Automated Synthesis, Placement and Routing of Large-Scale RSFQ Integrated Circuits

by
Jude Francois de Villiers

Thesis presented in partial fulfilment of the requirements for the degree of Master of Engineering (Electrical and Electronic) in the Faculty of Engineering at Stellenbosch University

Supervisor: Prof. C. Fourie
Electrical and Electronic Engineering
Stellenbosch University

March 2021

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Signature: J. de Villiers

Date: March 2021

Abstract

English

The dissertation presents the development and implementation of an automated synthesis tool for the synthesis of Rapid Single Flux Quanta (RSFQ) logic circuits from a high-level description of a circuit. With today's excessive demand for high-performance computation, the current complementary metal-oxide-semiconductor (CMOS) circuits already at its pinnacle does not deliver on present requirements, therefore a new form of integrated circuits will be highly-priced. RSFQ logic circuits may be a solution with new interest and funding it may become a reality in the future. The developed synthesis tool, ViPeR can create a 4-bit adder circuit that can operate at 35GHz. Further development to improve the tool and processes seems possible. ViPeR consists of a subset of tools, each with a unique purpose in synthesising the RSFQ circuit. The primary challenges were to balance the logic gate with delay flip-flops (DFFs), generating an optimal layout for the logic gates and finally to integrate a clock distribution network into the circuit. Two other tools were also created to aid in the verification and final chip generation of the circuit, namely Die2Sim and chipSmith. The synthesised circuits are evaluated and compared to existing circuits, paving the way for future development.

Afrikaans

Hierdie proefskrif handel oor die ontwikkeling en implementasie van 'n sintese-instrument vir die sintese van 'n *Rapid Single Flux Quanta* (RSFQ) logiese stroombaan, vanaf 'n hoëvlak beskrywing van so 'n stroombaan. In heedendaagse tyd is daar 'n uiters hoe aanvraag vir 'n hoë spoed rekaar berekeninge baie gesog. *Complementary metal-oxide-semiconductor* (CMOS) stroombane is die huidige oplossing maar die spoed hiervan is alreeds op maksimum punt, daarom is 'n nuwe vorm van geïntegreerde stroombane nodig. RSFQ logiese stroombane blyk om 'n oplossing te wees met onlangse belangstelling en befondsing kan dit 'n werklikheid word. Die ontwikkelde sintese-instrument, ViPeR kan 'n 4-bis-sommeerder skep wat teen 35GHz loop, verdere navorsing en verbetering blyk moontlik. ViPeR bestaan uit 'n stel instrumente wat elk 'n unieke doel het om die RSFQ-stroombaan te sintetiseer. Die primêre uitdagings was om die logiese hekke met *delay flip-flops* (DFFs) te balanseer, 'n optimale uitleg vir die logiese hekke te genereer en uiteindelik 'n klokverspreidingsnetwerk in die stroombaan te integreer. Twee ander instrumente is ook geskep om te help met die verifikasie en finale opwekking van die skyfies, naamlik Die2Sim en chipSmith. Die toets gesintetiseerde stroombane is geëvalueer en vergelyk met bestaande stroombane, met resultate wat die weg baan vir toekomstige ontwikkeling.

Acknowledgements

I want to thank the following people for helping me in the past 2 years. Firstly I would like to thank my supervisor Professor Coenrad J. Fourie for his continuous expert support and guidance whom without this dissertation will not be possible. The support from my friends who have helped me to keep motivated the past few years. I appreciate the support of my research colleagues who have helped me overcome any technical challenges. A special thanks to my family for their love and support throughout my life.

Contents

List of Figures

List of Tables

Nomenclature

Variables and functions

I_c	Critical current
T_C	Critical temperature
R	Resistance

List of Abbreviations

AQFP adiabatic quantum-flux-parametron

CDN clock distribution networks

CMOS complementary metal-oxide-semiconductor

CSV comma serperated value

DFF delay flip-flop

DRC design rule checking

EDA electronic design automation

GDSII Graphic Design System

HDL hardware description language

JTL Josephson transmission line

KSA Kogge–Stone adder

PTL passive transmission lines

PTLRX passive transmission lines receiver

PTLTX passive transmission lines transmitter

RSFQ Rapid Single Flux Quanta

SFQ Single Flux Quanta

SPICE Simulation Program with Integrated Circuit Emphasis

TCAD Technology Computer Aided Design

ViPeR Verilog to Placemenet and Routing

1 Introduction

1.1. Motivation

Modern-day computing is approaching its pinnacle rapidly and a substitute for CMOS is required if we want to maintain Moore's law. Superconducting computing promises to be our saving grace. Superconducting systems promise significant improvement on performance and requires a fraction of the power of semiconductors even when including cryogenic refrigeration [1].

The primary purpose of the research and development of superconducting electronics is to improve design time and reliability of complex circuits significantly. A user should be able to easily develop a superconducting circuit without having to require a deep understanding of the circuit's inner working. Complete electronic design automation (EDA) toolchains is essential to help a developer design superconducting circuits. This will allow for broader adoption of superconducting systems, thus drawing more interest into the field, which in turn will give the field the required boost to become mainstream. [2]

1.2. Background

Standard semiconducting materials' resistivity decreases with an increase in temperature. With superconducting materials, the resistance decrease with a drop in temperature. For a material to be in a superconducting state($R = 0\Omega$), it has to be below its critical temperature T_C otherwise it will be considered a normal conductor($R > 0\Omega$).

Superconducting electronics has been researched and available for years but did not receive much attention since conventional semiconducting electronics being sufficient. Several methodologies of superconducting logic circuit exist currently, the two main types are RSFQ and adiabatic quantum-flux-parametron (AQFP). The former having a significantly higher clock speed where the latter has better power consumption. Further on in the dissertation, only RSFQ circuit will be focused on as it is the method of choice for the Stellenbosch Coldflux team.

In 1962 the Josephson effect was discovered by a British physicist Brain Josephson which describes quantum tunnelling in superconductors. IBM later in the mid-1960s

used the effect to create a switch named the Josephson junction and then in the 1970s they envisioned to build a superconducting computer with it. In 1983 IBM ceased further research on their superconducting computer because CMOS at the time was dominating the field. At Moscow State University in 1985, researchers developed a new type of superconducting logic called RSFQ which utilised Josephson junctions in superconducting loops. [3]

In conventional semiconducting logic circuits, data is represented with DC voltage levels. With RSFQ, data is represented with a short pulse measured in picoseconds. Therefore binary data is represented with the presence or absence of a Single Flux Quanta (SFQ) pulse. [4]

Initial numerical simulations showed that an RSFQ gate could operate in excess of 300GHz [4]. Later on, a T-flip flop was developed and clocked to 770GHz [5]. Although a functional RSFQ CPU might not reach those exact speeds, it is significantly higher than modern-day CPUs can reach.

1.3. Objectives

The main objective of the dissertation is the development of a set of tools that will enable a user with an understanding of semiconducting circuits to create a complete superconducting logic circuit. The tools developed are ViPeR, Die2Sim and chipSmith. All the development of the tools was done in C++ for performance and versatility.

For the project to prove its worth, the following milestones are essential to ensure that the main objective is reached.

1. CMOS to RSFQ conversion: Manipulating a CMOS circuit described in hardware description language (HDL) to adhere to RSFQ requirements.
2. Circuit layout: Place the logic gates in an optimal layout ensuring successful routing of the interconnects.
3. Clocking and layout: Creating a balanced clock distribution network allowing the signals to reach all the logic gates within the same clock period.
4. Circuit verification: Develop a hassle-free tool enabling swift testing by simulating the circuit.
5. Final chip design: Generate a complete superconducting logic circuit adhering to chip manufacturing criteria.

6. Easy adaption of tools: The toolchain must be able to easily allow for new parameters(i.e. a new cell library) without modification of any algorithms or code.

1.4. Overview

Chapter 2 gives a basic overview of all the different aspects that are involved in having a fundamental understanding of RSFQ circuits. The chapter starts at giving an outline of Josephson junctions which forms the basis on which RSFQ circuits is built on and then focuses on the different aspects that are required to build and understand functional RSFQ circuits.

A brief overview is given of the external tools that are used in the dissertation is given in Chapter 3. The objective is to give an understanding of the tools and how they are integrated. The first tools looked at is ABC, which forms part of the initial logic synthesis of generating an RSFQ specific circuit. Next Qrouter which routes all the interconnects(tracks) between the gates efficiently. The final tool, JoSIM forms a critical stage in verifying that a circuit is working correctly.

The primary focus of the dissertation is in Chapter 4, which goes into depth of the circuit synthesis process. ViPeR was the tool created to house the whole synthesis process. The sequence of steps starts with converting a standard logic circuit in a HDL model into a RSFQ circuit by inserting splitter and DFF gates. All the gates have to be optimally placed to allow for a H-tree clock to be integrated. The process is completed by creating the final placement of the gate and having the tracks routed.

Chapter 5 looks at the implementation of the verification tool, Die2Sim. The primary goal is to get a simulation running using JoSIM of the developed circuit to ensure that it is operating correctly. Then Die2Sim handles the necessary associations of the gates and tracks while providing a test bench for the circuit to operate in.

Once everything is satisfactory with the synthesised circuit, it is placed onto a chip which is discussed in Chapter 6. The tool created, automates the process and handle all the chip overheads is named chipSmith. Chip overheads include creating fill around all the structures to fill up the voids and to create power(bias) connections to all the gates.

Chapter 7 investigates the performance and success of the circuit synthesis(ViPeR). Several circuits were created and tested to show how the algorithms handle different circuit sizes. All the tests were successful except for the biggest circuit that which had a minor timing miss-match.

2 Superconducting Circuits

2.1. Introductions

Superconducting electronics functions differently to standard semiconducting technology used today and required different thought process to understand. The focus of this chapter is to give a fundamental understanding of how RFSQ logic circuits operate. It starts by discussing how a Josephson junction operates and then steps through the levels.

2.2. Josephson Junction

The building blocks of RSFQ technology are Josephson junctions which are used as switching elements. The junctions utilised the Josephson effect, which describes a non-superconducting material sandwiched between two superconducting materials illustrated in Figure 2.1. The Josephson effect occurs when the current in one of the superconducting layers reaches the critical current I_C, the current then can flow through the insulating barrier without any resistance [6].

2.2.1. Physical Design

The Coldflux team utilises the SFQ5ee process by MIT Lincoln Laboratory to manufacture the wafers(chips). The process consists of nine superconducting layers with niobium being the superconductor. Table 2.1 summaries the different materials utilised to manufacture the wafers in the SFQ5ee process. A basic cross-section of a Josephson junction is illustrated in Figure 2.1, aluminium-oxide is used as the insulator and niobium being the superconductors. [7]

Material	Application
SiO_2	Interlayer dielectric
Nb	Superconducting metal
Mo	Resistive layer
AlO_X	Josephson junction insulator

Table 2.1: Different material layers in the SFQ5ee process [7]

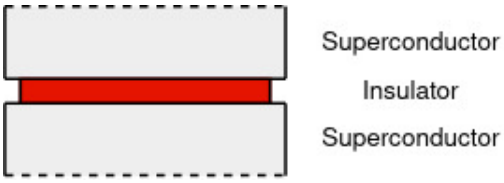

Figure 2.1: A basic cross section of a JJ

The manufacturing process of a superconducting wafer is imperfect, causing the structures' geometry to be malformed. Technology Computer Aided Design (TCAD) tools can create realistic models structures within a wafer in order to help check for any potential issues. Figure 2.2 and 2.3 illustrates an accurate representation of how a manufactured SFQ5ee Josephson junctions looks like using the TCAD tool Katana. The different metal layers and etched edges can clearly be seen in Figure 2.2 along with the ground plane the junction sits on. Figure 2.3 illustrates the dipping of the vias in the junctions. [8]

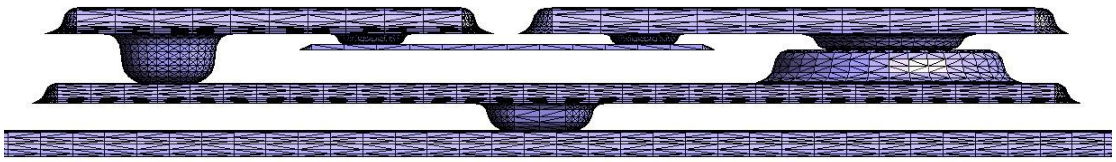

Figure 2.2: A side view of a TCAD rendering of a Josephson junction [8]

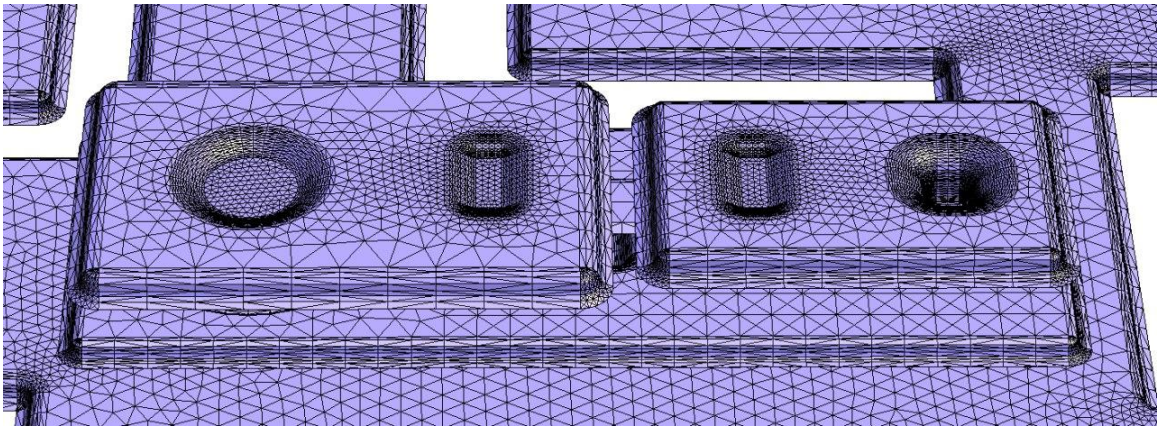

Figure 2.3: A top down view of a TCAD rendering of a Josephson junction [8]

2.2.2. Application

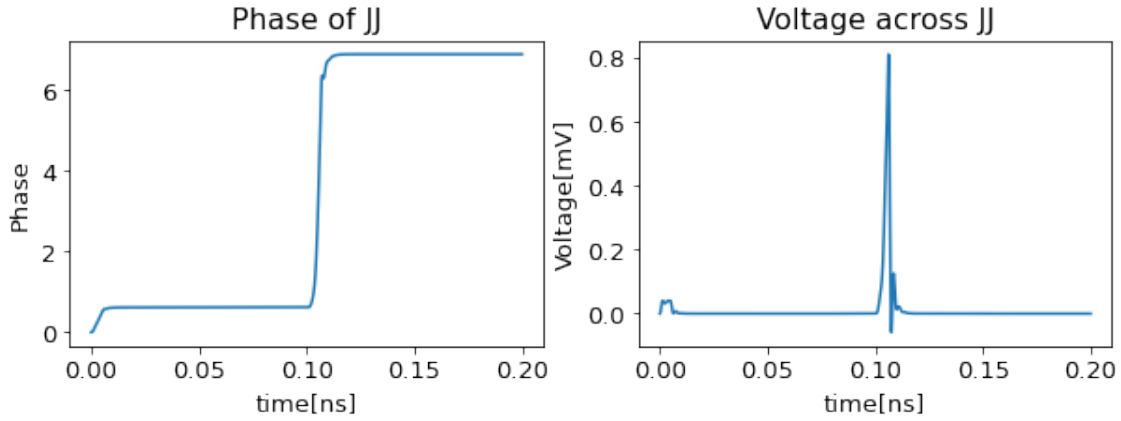

Figure 2.4: JoSIM simulation of a JJ undergoing phase change and the equivalent voltage spike

When a pulse is passed through a junction, it induces a quantised leap of the Josephson phase ϕ inducing $\Delta\phi = 2\pi$ as depicted in Figure 2.4. A standard pendulum model represented in Figure 2.5 helps to describe a Josephson junction. The junction is biased with a DC current $I_{bias} \leq I_c$ which corresponds to a (critical) applied torque to the pendulum ensuring that $\phi > 0$. By doing so allows a pulse to push $\phi > \pi$ forcing the pendulum to complete a full rotation. [4]

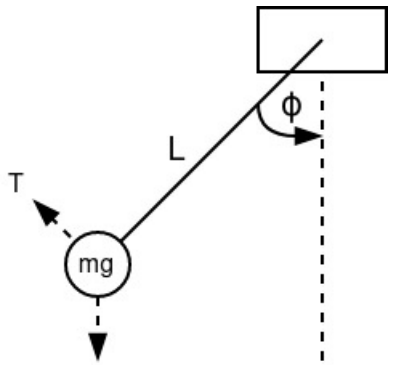

Figure 2.5: Pendulum model used to help describe JJ

2.3. RSFQ Gate Operation

RSFQ gates have a few unique features which cause it to operate differently to standard CMOS gates. The first few differences are illustrated in Figure 2.6, namely the number of connections an output pin can have to different input pins(fanout) and the RSFQ gate requires an additional input pin. Another difference is how the data(signals) is represented as mentioned in Section 1.2 and illustrated in Figure 2.7 [9].

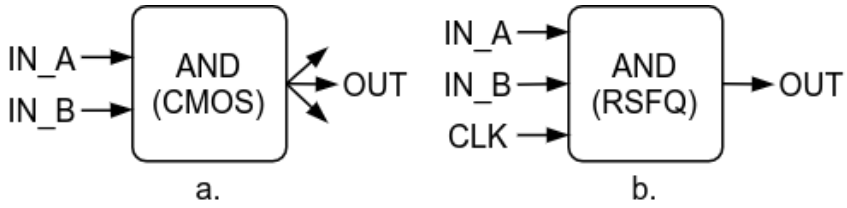

Figure 2.6: CMOS(a) and RSFQ(b) logic block

The first difference is that RSFQ gates require an additional input pin and that being a clocking input. It allows for the gate to only execute its operation when the clock signal has been received. For the gate to operate correctly, the input signals must be received before the clock signal arrives. Figure 2.7(b) demonstrates this and the resulting signal is only produced a time after the clock input has been received. Having to store input SFQ pulses until the pulse from the clock arrives creates a delay when compared to a CMOS gate which produces a result instantly.

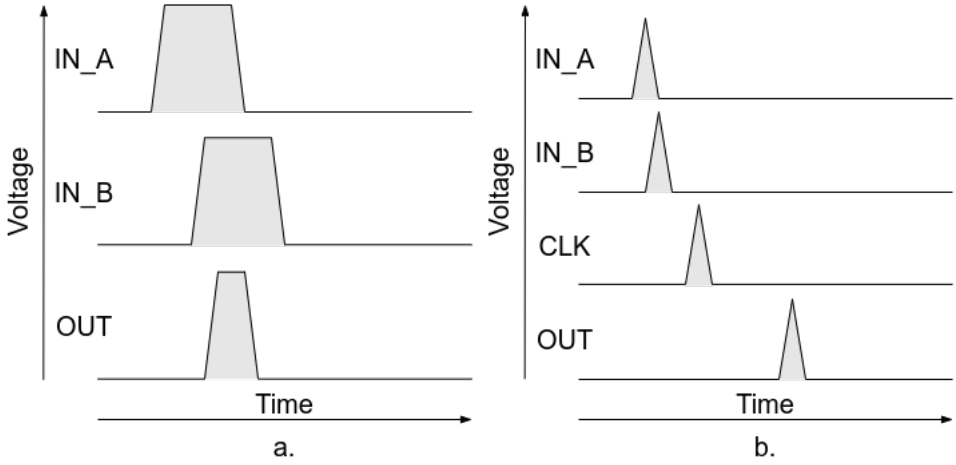

Figure 2.7: How CMOS(a) and RSFQ(b) logic compare using voltage levels and SFQ pulse respectively.

A limitation of RSFQ gates is that an output pin can only be connected to a single input pin whereas with a standard semiconducting gate's output pins can have multiple connections. To overcome this, an unique RSFQ gate allowing for multiple output connections has to be utilised in conjunction with the operational RSFQ gates.

2.4. Interconnects

Different SFQ circuits can be connected using a conductor(inductor), but the length of interconnects is limited to short distances [9]. To overcome this limitation, special circuitry is used to propagate the SFQ pulses over greater distances. Two different methodologies can be utilised as interconnects for SFQ circuits, namely passive transmission lines (PTL) and Josephson transmission line (JTL).

A JTL is essentially a pulse regeneration circuit that utilised active Josephson junctions to propagate the signal further to its destination with no signal deterioration. The main drawbacks of JTLs are that the SFQ pulse propagation speed is 5 to 10 times slower compared to PTLs and it requires significantly more power due to it requiring a DC bias current. [10]

PTLs are similar to coaxial cables as depicted in Figure 2.8 and are purely passive components due to not requiring a DC bias current. The SFQ pulse propagation speed is calculated to be 95.1μm/ps using MIT's SFQ5ee process [11]. The main drawback from using PTLs is that they have poorly matched impedance with the RSFQ circuits which cause signal reflections leading to complications. In order to mitigate impedance mismatch, PTL drivers and receivers are incorporated into the circuit to ensure that the impedance match [10]. Another advantage that has the PTL driver and receiver integrated into the RSFQ gates allow for row-based layout and routing [12].

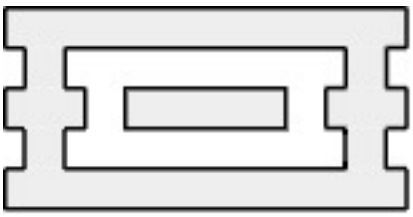

Figure 2.8: Cross section of an PTL

2.5. RSFQ Logic Synthesis

In order to be able to convert a traditional logic circuit into an RSFQ circuit, the drawbacks mentioned in Section 2.3 must be overcome. Due to all the operational gates being clocked, DFFs have to be incorporated into the circuit. The final obstacle is to account for the gates only having a fanout of one.

The first requirement for the conversion of the circuit is to ensure that the timing of the SFQ pulse trigger the gates are at the correct interval. Every boolean operational RSFQ gate is clocked to ensure so that the circuit is balanced, so there are no gates miss triggering. For the circuit to be balanced, the clock levels must first be calculated and then DFF are inserted appropriately to balance the circuit. Figure 2.9 and 2.10 illustrates how the DFFs are utilised to balance the circuit.

Most RSFQ gates have only an output fanout of one, preventing it from being connected to multiple gate inputs. To overcome this, a splitter gate is utilised that has a

fanout of two. Once all the necessary changes have been made to the circuit, the splitters are inserted where a gate's output has multiple connections. Figure 2.10 illustrates the placement of the splitter gates in a full adder circuit where all the gates have multiple output connections. The splitter gate is designed to have little as possible delay, which ensures that it will have an insignificant delay to the SFQ pulse propagation. Having the delay insignificantly small prevents it from requiring its own clock level, which increases the circuits operating speed and reduces the complexity of the overall circuit.

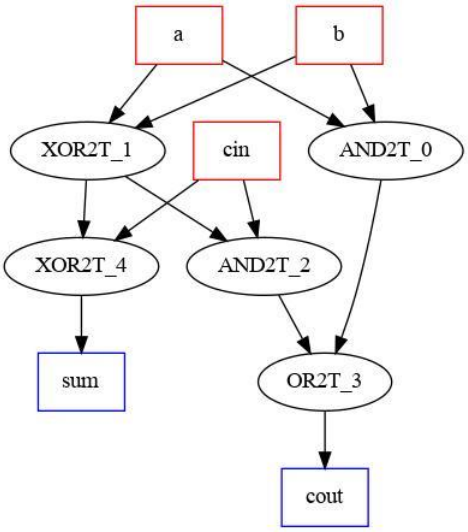

Figure 2.9: Full adder logic flow for CMOS circuit

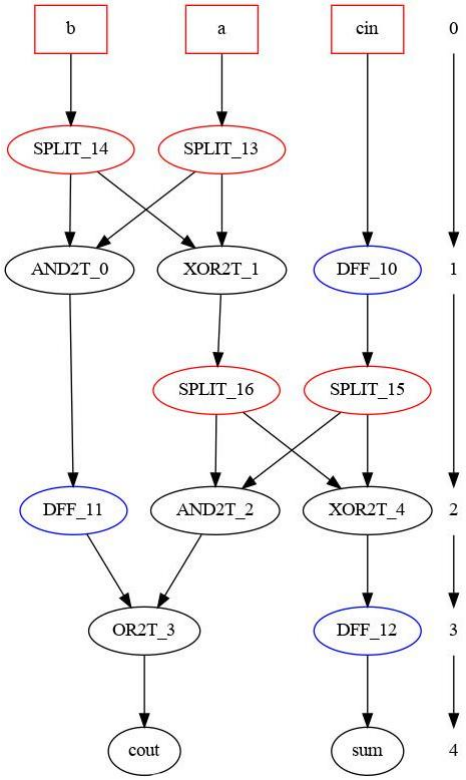

Figure 2.10: Full adder logic flow for RSFQ circuit

2.6. Clock Distribution Network

Most gates in an SFQ circuit needs a clock signal as stated in Section 2.5 therefore a clock distribution networks (CDN) is required. Several papers [4, 13–15] have suggested different types of clocking schemes. The H-tree clocking scheme is still the preferred scheme for large scale circuits [9].

The H-tree structure is superior since it ensures the clock is balanced (the clock signals reach all the gates at the same time), ease of implementing and it distributes that signal equally throughout the circuit. Figure 2.11 illustrates the basic layout of a H-tree. The initial clock signal is injected into the centre splitter. Then the signal(SFQ pulses) is

evenly distributed throughout the whole circuit until there is enough SFQ pulses for all the gates. With the splitter gate having a fanout of two, it suffices perfectly with the H-tree CDN scheme.

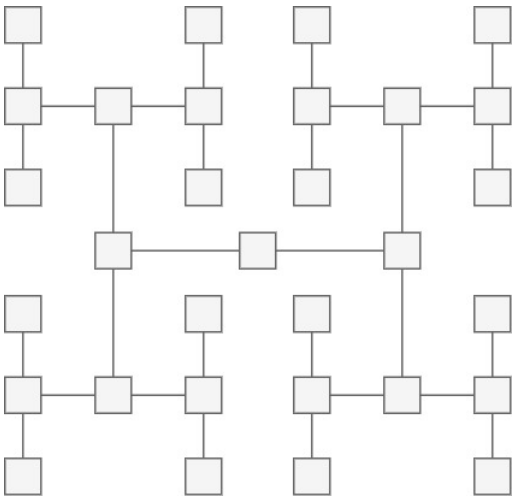

Figure 2.11: H-tree

2.7. Shielding

An unique problem to superconducting electronics is that the physical circuit is sensitive to external magnetic fields. Magnetic fields couple themselves onto the circuit structures which degrades performance or can prevent operability of the superconducting circuit. In order to mitigate external magnetic interference, shielding and moats are utilised in the circuit. [16, 17]

There are two schemes of shielding that are employed, off-chip and on-chip. Off-chip shielding utilises a high permeability ferromagnetic material which encases the wafer in order to diverge the magnetic flux around it. On-chip shielding is when a sky plane layer is employed to add another layer of shielding which exploits Meissner effect to ensure the shielding is optimal. [18, 19]

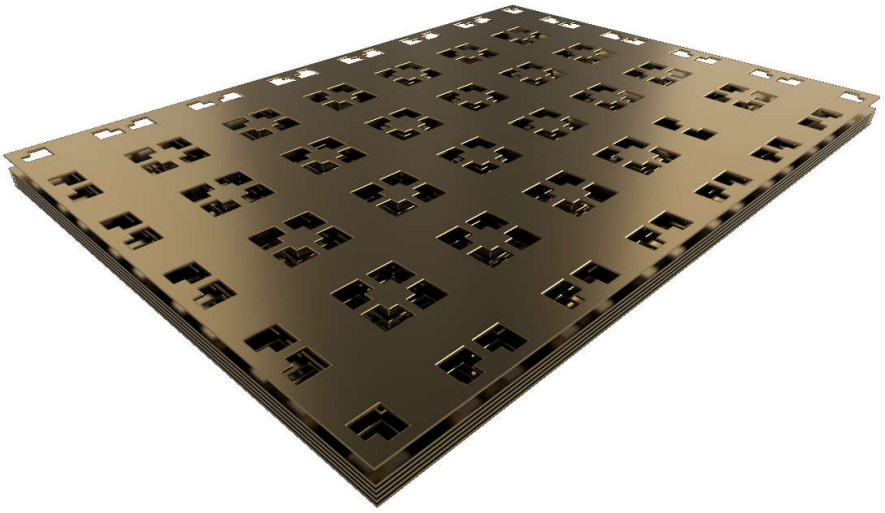

Figure 2.12: 3D Render of a splitter cell demonstrating the shielding layer [8]

Another technique employed to diverge residual magnetic fields is the installation of moats. It is openings in the ground plane used to attract fields away from critical parts of the circuit and for it to be captured in the hole [16]. Figure 2.12 illustrates the moats in the sky plane [8].

2.8. Summary

To grasp a decent understanding of how superconducting circuits operates, vast knowledge of the field is required. For the circuits to operate successfully, many aspects and rules must be considered and adhered to. Much research can still be done in most aspects of superconducting electronics to give a better, more concrete understanding of the field. Current development relies heavily on academic teams to effectively design large scale circuits.

3 Utilisation of CAD Tools

3.1. Introductions

A collection of EDA tools is essential in developing large scale superconducting circuits. The system is far too complex to be handled in a single software package, so a subset of tools is utilised. The goal of the tools is to speed up the generation of RSFQ circuits, which would be too tedious to done by hand. This chapter will give an overview of the required tools that a designer will need to aid the development of superconducting circuits such as synthesis of logic circuits, placement and routing of gates along with simulations tools.

3.2. Logic Synthesis

The logic synthesis purpose is to convert a logic circuit that is human-generated(readable) to a gate-level description. The circuit described in a high-level of abstraction enables the designer to effortlessly create a logic circuit without being concerned about the circuits inner workings. The process starts with the circuit being described in HDL(Verilog) format. The tool then processes the HDL circuit and outputs the gate representation of the circuit in a BLIF(Berkeley Logic Interchange Format) [20] file. Once the described circuit is defined on a gate-level, the superconducting electronics specific features can be incorporated [9]. Section 2.5 described the process for enabling a logic circuit to adhere to the RSFQ requirements.

The tool utilised for the logic synthesis is ABC [21] which is developed by UC Berkeley. ABC primary use is to optimally convert the behavioural description of the circuit to a gate-level description of the circuit. Another logic synthesis tool available is Yosys [22], which is based off ABC.

3.3. Placement and Routing

Placement and routing is the aspect of superconducting electronics that is continuously evolving and an optimal algorithm is yet to be developed. Routing of the interconnects is mostly dependent on the position of the gates due to it being easier to route if the connections can be kept as short as possible. The main criteria for optimal placement

of RSFQ gates are low track length variance, track intersections(via count) kept to a minimal and cell distribution must be equally balanced as possible to improve congestion.

A highly regarded placement algorithm is SimPL which outperforms older placers [23]. SimPL has been implemented in the RSFQ field, but it does not have the highest quality placement in terms of track length which is of significant importance [24].

Routing all the gates' pins together is critical however, the placement of the cells severely influences the route-ability of the interconnects. The most widely adopted routing tool Qrouter [25] is proven to be effective with routing RSFQ circuits. Qrouter allows for a cost functions which enable the router to favour specific track characteristics.

3.4. Circuit Verification

Circuit verification is an essential step in developing superconducting circuits. Two methods can be utilised for verifying circuits, logic or electrical simulation. Logic simulators being significantly faster and the electrical simulation slower but is more precise.

Logic simulating is favoured due to the fast execution speed of the simulation. The whole process starts by obtaining the Verilog timing files of the cells using a software package TimeEX [26]. iVerilog [27] is then utilised to run the simulation [18].

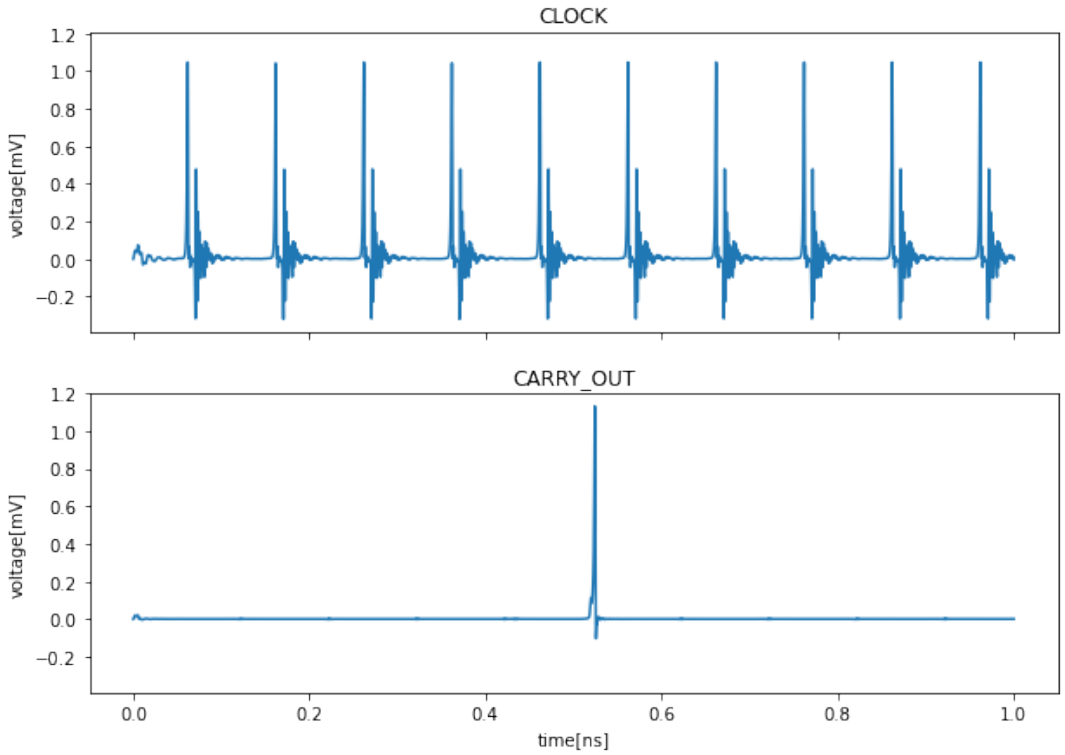

Figure 3.1: JoSIM simulation of a full adder RSFQ circuit

Electrical simulation of the designed circuits is a critical stage in the development of the circuit. Simulating the superconducting circuit is considerably different from the common semiconducting Simulation Program with Integrated Circuit Emphasis (SPICE) simulations due to the lack of Josephson junctions elements. Several superconducting SPICE engines have been developed in the past namely WRspice [28], JSIM [29] and PSCAN [30]. JoSIM [31] has superseded the rest due to the improvements in simulation speed and expandability. The results from JoSIM is provided in a comma serperated value (CSV) format which can be viewed as a graph. Figure 3.1 represents the results from a JoSIM simulation of a full adder circuit.

3.5. Summary

EDA tools are essential at aiding the developer to create superconducting logic circuit tools. Currently, far too many components of the existing toolchain have to be modified to cater for superconducting field specifics. Thus to speed up the development process of the superconducting electronic field, more tools need to be developed and published.

4 Circuit Synthesis - ViPeR

4.1. Introduction

The focus of this chapter will be to expand on the implementation and development of the circuit synthesis tool, Verilog to Placemenet and Routing (ViPeR). ViPeR is a collection of various tools each with a different purpose in the process of synthesising an RSFQ circuit. The main goal of ViPeR is to be able to process a HDL file(Verilog) and to produce a complete working RSFQ circuit of the described circuit. The primary steps are the implementation of SFQ logic specific features, placement of cells, gate clocking and routing of interconnects in between gates. All the software and algorithms development is implemented in C++ for performance and compatibility reasons.

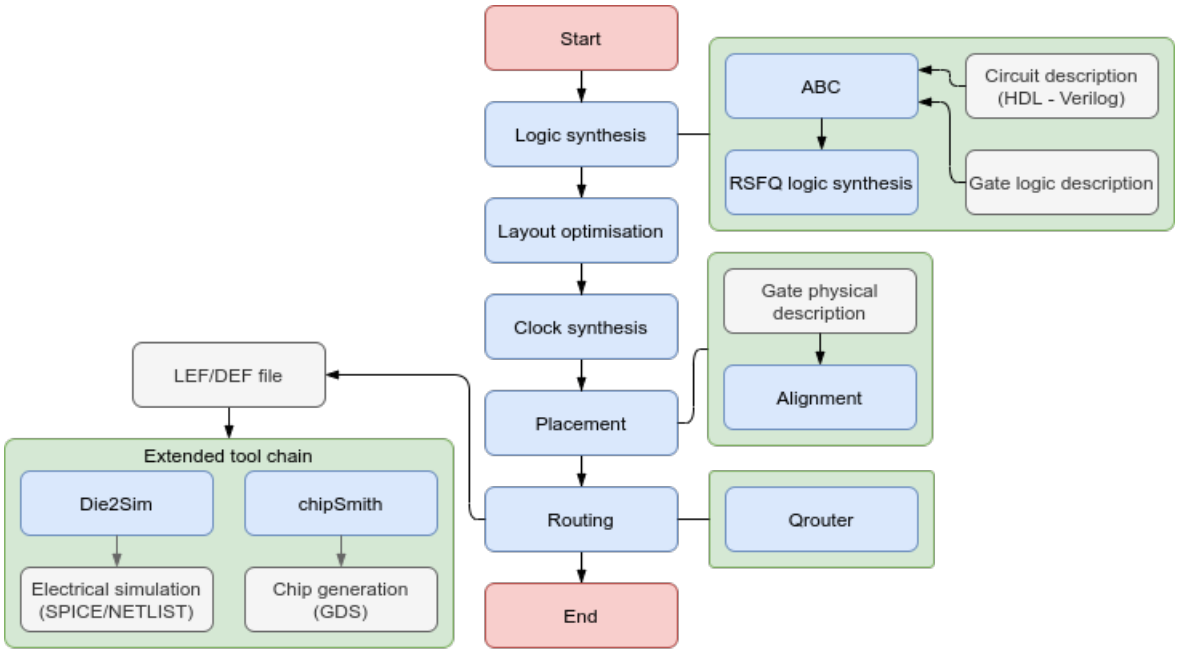

Figure 4.1: Flow diagram of ViPeR

The complete EDA tool flow is portrayed in Figure 4.1. The flow diagram shows the different processes and sub-processes of the synthesis procedure. After the circuit synthesis is completed, the extended toolchain is also depicted in the flow diagram giving a complete picture of the whole process until the final design.

4.2. RSFQ Logic Synthesis

The process of generating an RSFQ circuit from a high level abstracted semiconducting circuit consists of three steps. It starts off with converting the described circuit into a low-level logic description, then the circuit's gates need to be balanced, followed by correcting the fanout of the RSFQ gates. Then fanout correction process is completed last otherwise it must be repeated.

4.2.1. Circuit Abstraction

ABC [21] tool from Berkeley provides a collection of tools enabling designers to generate gate-level descriptions of circuits from a high-level of abstraction. ABC allows for easy modification of its algorithms to enable designers to customise it for their needs. With the implementation of ViPeR, no modifications were made to ABC and all the RSFQ specific features were done post-processing in ViPeR.

ABC interprets the described circuit in Verilog into a gate-level description in a BLIF [20] file format. BLIF file was designed to describe circuits at a logic(gate)-level and is used throughout ViPeR to transfer circuit information. A full adder described in Verilog is depicted in Listing 4.1 and the BLIF file is provided in Apendix B.

```
module FullAdder(a, b, cin,
   cout, sum);
 input a, b, cin;
 output cout, sum;

 assign sum  = a ^ b ^ cin;
 assign cout = ((a ^ b) & cin)
    | (a & b);
endmodule
```

Listing 4.1: Verilog code for a full adder circuit

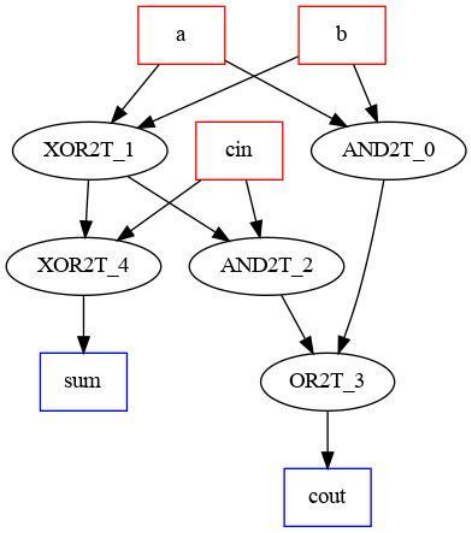

Figure 4.2: Full adder logic flow

Within ViPeR, a tool was created to visualise the flow of a circuit easily. The tool iterates through the gates and nets starting from the inputs and flows to the outputs the circuit. The circuit is then described in a dot language file [32] and then parsed to Graphviz [33] which creates a flow diagram. Appendix C gives an example of a full adder

circuit being described in the dot language and Figure 4.2 is the generated flow diagram from Graphviz of the full adder.

For ABC to operate, it requires a description of the different gates that are available in order for it to synthesise any circuits. Appendix A provides the configuration file which describes the ColdFlux RSFQ cell library [34] which is utilised in ViPeR. Only the operational gates are described due to the other gates been RSFQ specific features is not normally handled by ABC.

4.2.2. Circuit Balancing

One of the characteristics of RSFQ circuits is that all the gates must be clocked. The gates being clocked creates a rather tricky challenge due to flow of the SFQ pulses through the circuit. Therefore path balancing is required. The opted method to make the implementation easier is to observe all the possible paths the signals can flow. For the circuit to be considered balanced, the path length and the gates must appear on the same level. Listing 4.2 illustrates all the different signal paths in an unbalanced full adder. To balance the circuit, DFFs are strategically inserted. There are 3 different stages where the DFFs are inserted, the process starts at the input side of the circuit and ends at the outputs.

```
1  a   -> AND2T_0 -> OR2T_3  -> cout
2  a   -> XOR2T_1 -> AND2T_2 -> OR2T_3 -> cout
3  a   -> XOR2T_1 -> XOR2T_4 -> sum
4  b   -> AND2T_0 -> OR2T_3  -> cout
5  b   -> XOR2T_1 -> AND2T_2 -> OR2T_3 -> cout
6  b   -> XOR2T_1 -> XOR2T_4 -> sum
7  cin -> AND2T_2 -> OR2T_3  -> cout
8  cin -> XOR2T_4 -> sum
```

Listing 4.2: All possbile routes a SFQ pulse can flow in an unbalanced full adder

The algorithm that determines all the possible routes as in Listing 4.2 and 4.3 is utilised repeated throughout the balancing process. The technique utilised is a recursive function that traverses through the gates using depth-first search. Figure 4.3 demonstrates the algorithm to determine the routes. Depth-first search allows for the clock levels of each operational logic gate to be determined. With an unbalanced circuit, the gates will appear

at different clock levels due to the gates appearing at different stages of all routes. When the circuit is balanced, all the gates must occur at the same depth in all the routes. Only once the circuit is balanced can the RSFQ circuit successfully operate. The algorithm is also utilised when calculating the optimal layout in Section 4.3.

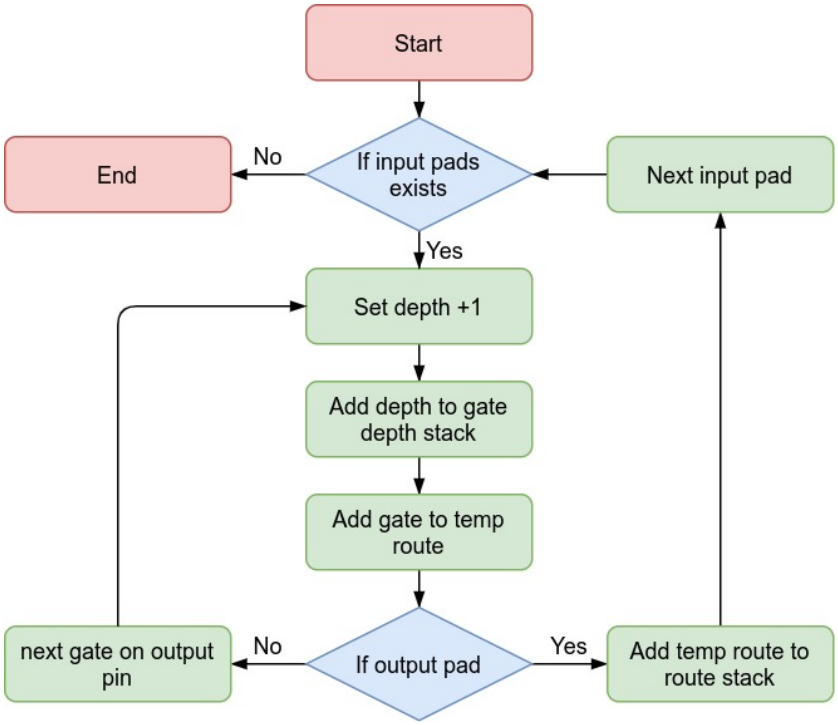

Figure 4.3: Flow diagram to find all possible routes.

The first step is to ensure that all the inputs pins are in sync. That entails that the SFQ pulse must reach the first common gate at the same time for all the inputs. The obvious solution will be to look at all the possible routes and compare all the routes to find the first common gate, but this can be tedious and can take unnecessarily long to calculate. For the optimal solution, it was determined that it is significantly easier to instead look at the last common gate which will occur at the end of the overall longest route. The implementation was made even easier by taking the longest route of each input pin and then the difference is calculated from the longest route. The calculated difference is used to insert DFFs right after the input pin. This ensures that the route to the first(and last) common gate is the same length.

Once the input DFFs have been inserted, the levelling of the operational logic gates can be balanced. The procedure starts by observing the different clock levels of each gate by iterating through all possible routes and determining the level it appears in that route. The difference in clock levels is then taken of the greatest and smallest clock level of each gate and the difference is inserted in DFFs after the gate to balance out the circuit. The clock levels are recalculated every time after a gate is balanced to ensure that an excess of DFFs is not inserted, causing the circuits to be unbalanced again.

The final step in balancing the circuit is to ensure that the resulting SFQ pulses of the circuit arrive at output pins at the same time. To ensure this, all the possible routes of the circuit must be the same length. The optimal solution is to look at the overall longest route and the longest route of each input pin and to add the difference in DFFs before the output pin.

```
1  a   -> AND2T_0 -> DFF_11  -> OR2T_3 -> cout
2  a   -> XOR2T_1 -> AND2T_2 -> OR2T_3 -> cout
3  a   -> XOR2T_1 -> XOR2T_4 -> DFF_12 -> sum
4  b   -> AND2T_0 -> DFF_11  -> OR2T_3 -> cout
5  b   -> XOR2T_1 -> AND2T_2 -> OR2T_3 -> cout
6  b   -> XOR2T_1 -> XOR2T_4 -> DFF_12 -> sum
7  cin -> DFF_10  -> AND2T_2 -> OR2T_3 -> cout
8  cin -> DFF_10  -> XOR2T_4 -> DFF_12 -> sum
```

Listing 4.3: All possbile routes a SFQ pulse can flow in a balanced full adder

Once the three balancing steps are completed, all the routes should have the same length. Specific logic circuits are better suited for RSFQ design by having shorter routes which ensure that fewer DFFs have to be inserted, improving overall power consumption, layout size and clock speed. The main drawback of the method used is that it only allows for combinational and not sequential(feedback) logic. If a sequential logic circuit is attempted, the recursive depth-first search will fall into an infinite loop preventing any further logic synthesis.

4.2.3. Fanning Out

A standard RSFQ operational logic cell only has a fanout of one. Therefore the output on a logic gate can only be connected to a single gate severely limiting the possible combinations of logic. The solution is to insert a splitter cell that has a fanout of two, allowing the output of a gate to be connected to more than one gates' input. The splitter gate does not have a clock input and has a fast operating time enabling the gate to have a minimal influence on the circuit's clock speed.

$$n_{depth} = roundup\left(\frac{log(n)}{log(2)}\right) \tag{4.1}$$

With larger circuits, the chance that a logic gate needs a fanout higher than two is significant. The limit can be exceeded when a typical binary tree structure of splitters is utilised to increase the fanout exponentially with the base of two. An iterative function is implemented over a recursive function with the result of increased control over the physical layout. The algorithm creates the tree, layer for layer. To determine the required depth of the tree, Equation 4.1 is utilised where n is the number of outputs that are required. Figure 4.4 illustrates an example of how the binary tree of splitters is constructed.

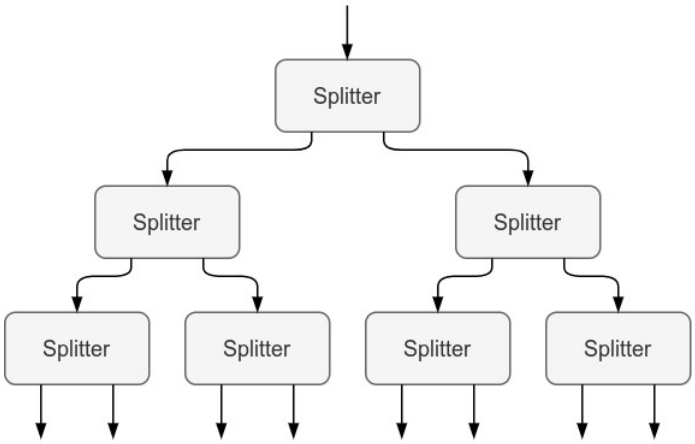

Figure 4.4: Binary tree of splitters.

With a binary tree size increasing exponentially, it can cause splitters to become freestanding(unutilised) if the number of required outputs is not of the base of two. The drawbacks of having freestanding splitter gates are that it physically takes up space on the chip and draws power unnecessarily.

JoSIM simulation was conducted to check what are the effects of unconnected outputs of splitter gate. The simulation results and the test bench code that as used is in Appendix D. The results show that the unconnected(grounded) output does not affect operation of the gate. Therefore, the implementation of the splitter binary tree to increase the fanout of greater than two is sufficient not to cause any unforeseen issues to the circuit.

4.3. Optimal Cell Layout

The layout of the gates is proven to be a significant factor in the performance of the logic circuit. Poor layout can cause the PTLs to be longer and having greater variance in length. It can lead to more complex(more vias and corners) PTLs than necessary making the delay longer and a higher chance for timing errors which all can lead to unexpected behaviour and degrade in performance. The layout scheme

follows a row-base cell placement system due to the flowing nature of the RSFQ circuits and the gates are optimised for it [12, 35]. The optimal layout of the gates relative to each other must first be determined before the clock synthesis and final placement is done.

The layout of the gates is vital to the success of the circuit. The opted method is a route based stacking optimisation. It has significantly fast execution time and works well with mid-scale circuits. For a full adder and 4-bit KSA circuit, the optimal layout is calculated in $87us$ and $900us$, respectively. The method works in two parts, optimally sorting all the possible routes and then stacking them together. The method used assumes that all the different gate sizes are similar.

4.3.1. Sorting

Most types of combinational logic involve two numbers that must be manipulated together. Arithmetic logic circuits are designed to enable the layout of logic circuits to flow smoothly and have few interconnects that cross over each other. Figure 4.5 demonstrates the addition of two numbers for example. With addition arithmetic, only the number in the same vertical position influences each other and the carryover from the previous operation.

		4	3	2	1
b		1	6	1	8
a	+	2	9	9	7

Figure 4.5: Addition arithmetic

+	b4	a4	b3	a3	b2	a2	b1	a1

Figure 4.6: Addition arithmetic in a single row

The starting(input) row for a combinational logic arithmetic sequence can therefore be generalised. The inputs of the two numbers can be alternated in chronicle order to help obtain the optimal layout for the logic circuit. Figure 4.6 depicts an example of the optimal order of the inputs. The secondary parameter for the routes to be sort by is the order of outputs. The same sorting principle is to be followed as with the order of the input pins. The order of the output pins must also follow the same order of significance as the input pins.

By having the layout of the input and output pins in order as described above, the routes utilised in Section 4.2.2 can be sorted accordingly. By having the routes sorted, the common gates in the route are more likely to be closer together to each other creating hotspots of the gates. It ensures that the interconnects are kept as short as possible.

4.3.2. Stacking

Once all the routes have been sorted accordingly, the routes are then stacked together into their final layout. With the layout scheme being row-based, the input pins of the circuit are at the top with the output pins at the bottom. The stacking process iterates through the sorted routes and packs(stacks) the route into the layout. If the gate has already been stacked, it is omitted. The process continues until all the gates in the layout is complete. Figure 4.7 shows the optimised layout of a full adder using the routes from Listing 4.3.

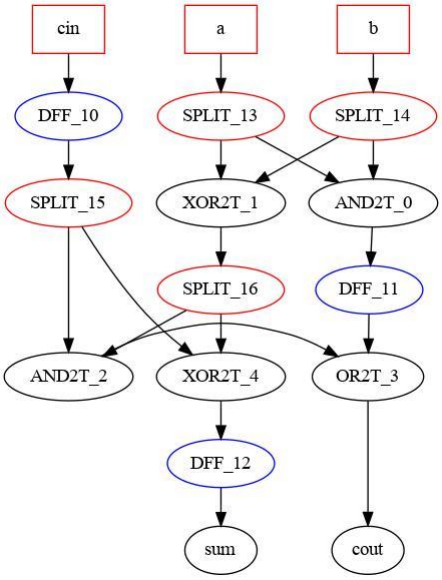

Figure 4.7: Layout of full adder circuit.

Once the routes are in the optimal layout, it is essential to note that rows do not represent clock levels anymore due to the addition of the splitter gates. Due to the splitter gates not requiring clock signal, the circuit can become longer than initially and the circuit may thin out(fewer gates) towards the output pins.

4.4. Clock Synthesis

Distributing the clock throughout the circuit is a crucial element of RSFQ circuits due to all the logic gates requiring a clock input. With RSFQ circuits having a fanout of one, a network of splitter gates is required to distribute the signal through the circuit. The primary goal of the clock distribution network is for it to be balanced as much as possible, enabling the gates to activate at the same time. If the clock network is too unbalanced, the gates can start miss firing which can cause the logic to give unexpected results.

An H-tree layout is the most common way to distribute a clock signal throughout an RSFQ circuit. It allows for optimal distribution due to the nature of the structure shape. The main drawback is to integrate the H-tree into a predefined circuit effectively. To overcome the integration drawback, a modified version of an H-tree is utilised which adheres to the distribution properties but has an alternative layout. Due to the layout of the circuit being row-base, it enables the use of inserting parts of the clock network into its own rows. The placement of the clock splitters do not necessarily have to be perfect, minor deviations are allowed.

Creating the clock distribution network is a multistep process, with a bottom-up approach. The whole clocking process starts by taking the logic gates and connecting them to the bottom end of the clock network. The clock network is then balanced twice followed by cleaning up the excess clock splitters. Once the whole clock network is created, it is then merged with the logic gates layout to complete the circuit.

4.4.1. Clocking the Logic Gates

The first step in the bottom-up clocking process is to connect the logic gates that require a clock signal to the initial clock splitter. With the layout of the circuit being row-based, a row of clock splitters can be inserted in between the rows of logic gates. The width of the splitter cell is smaller than the operational logic gates and having a fanout of two, making it optimal to distribute the clock signal for the logic back above and below the clock row. Figure 4.8 illustrates the layout for how the clock splitters are connected to the logic gates. The blue and green block represents the clock splitters and the logic gates, respectively. Distributing the clock signal to the row above and below helps reduce the number of clock rows needed and helps improve the layout density.

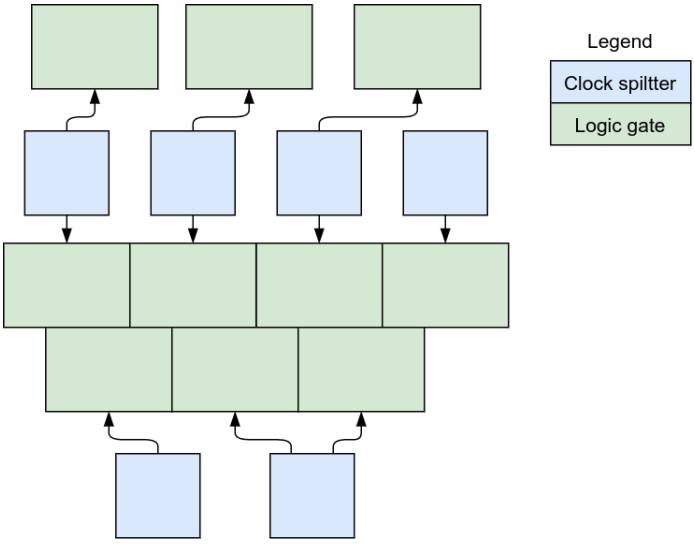

Figure 4.8: Clock distribution network connection to the logic gates

4.4.2. Balancing the Clocking Row

Balancing the row of clocking splitters in Section 4.4.1 is the second step in the clock synthesis process. The goal is to allow for 1 input clock signal to be distributed to a single row of the clock splitters while ensuring the distribution is balanced. Figure 4.9 shows the tree structure of the clock distribution and Figure 4.10 illustrates how the tree structure is compressed into a single row. Compressing the tree vertically ensures that the structure is balanced with all the different levels being equally distant from each other.

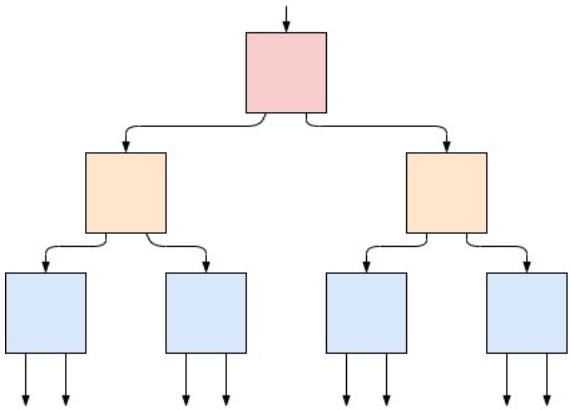

Figure 4.9: The clock binary tree for a row of clock splitters

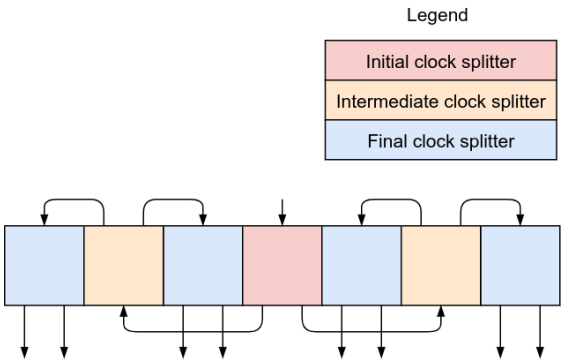

Figure 4.10: Balanced row of clock splitters

The approach is to distribute the clock signal following a binary tree scheme. The process starts by calculating the required amount of levels needed using Equation 4.1 where n is the number of clock splitters that is connected to the logic gates in a row. The generation of the tree starts at the first clock splitter, then at every output of the splitter another splitter is added until the desired(calculated) depth is reached. With every level of clock splitters that are accumulated, the splitters are inserted into the into alternating(odd number) positions to keep the clock network balance and to have the clock signal evenly distributed to all the logic gates. Figure 4.10 portrays how the row of splitters will be stacked together.

The nature of a balanced binary tree is that the final nodes(splitter gates) always amount to 2^x. Therefore, there is a high probability that there is going to be freestanding clock splitters due to the number of gates requiring a clock signal is $\leq 2^x$. The freestanding clock splitters do not have a significant negative effect expect for power usage and chip real estate, Section 4.2.3 stated that there are no unforeseen electrical effects. Once the clock network is completely generated, the clock is cleaned up of excess clock splitters with a post-processing function explained in Section 4.4.4.

4.4.3. Main Clock Distribution

The main clock network must take the single clock signal input and must equally distribute it to all the clock rows described in Section 4.4.2. In order to keep the clock network balanced, the insertion of main clock gates needs to be vertical(column) because the previous clocks are horizontal. The generation of the main clock network almost follows the same procedure as with the clock splitters in Section 4.4.2 with the exception that the final nodes(splitters) of the main clock tree is connected to the input of the clock distribution rows. Figure 4.11 illustrates how the main clock network on the left equally distributes the clock to each of the clocking rows.

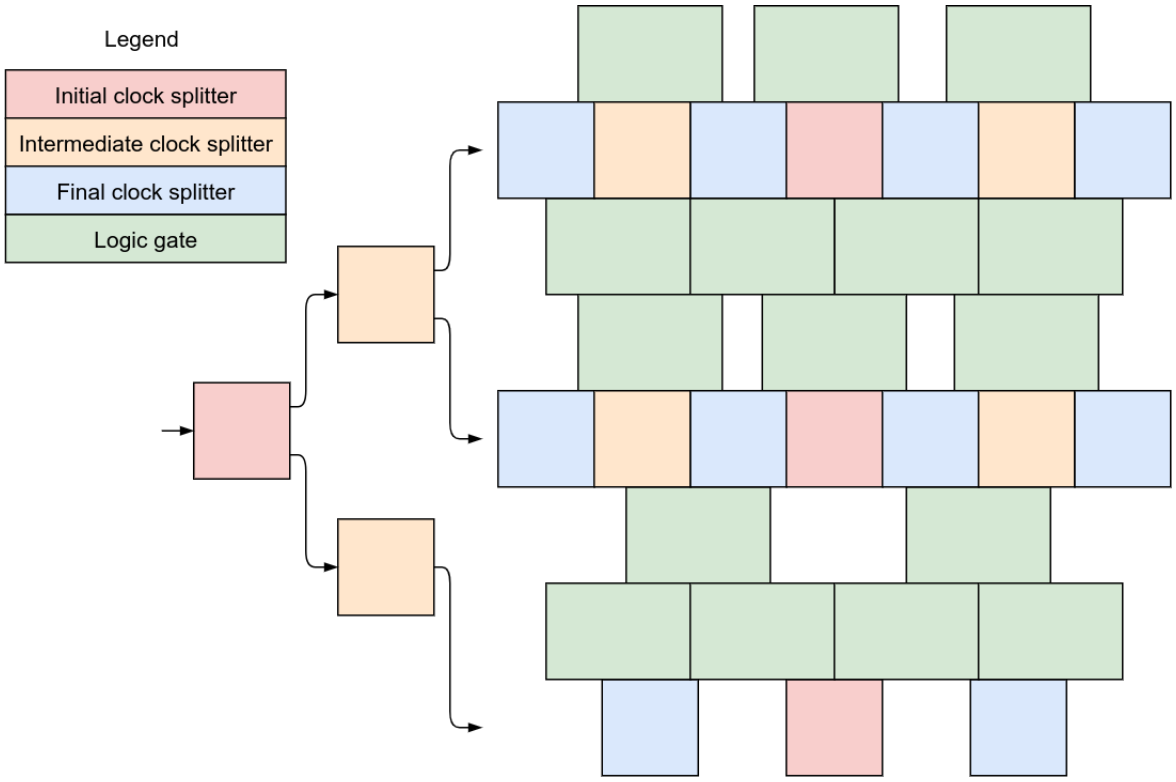

Figure 4.11: Main clock distribution network

Placement of the main clock network in the layout is done by stacking(inserting) the clock splitters on the left side of the gates. The vertical clock gates are equally distributed to the nearest row of gates(clock splitter and logic gates) and then inserted. Figure 4.12 demonstrates the insertion of the main clock network into the same circuit from Figure 4.11.

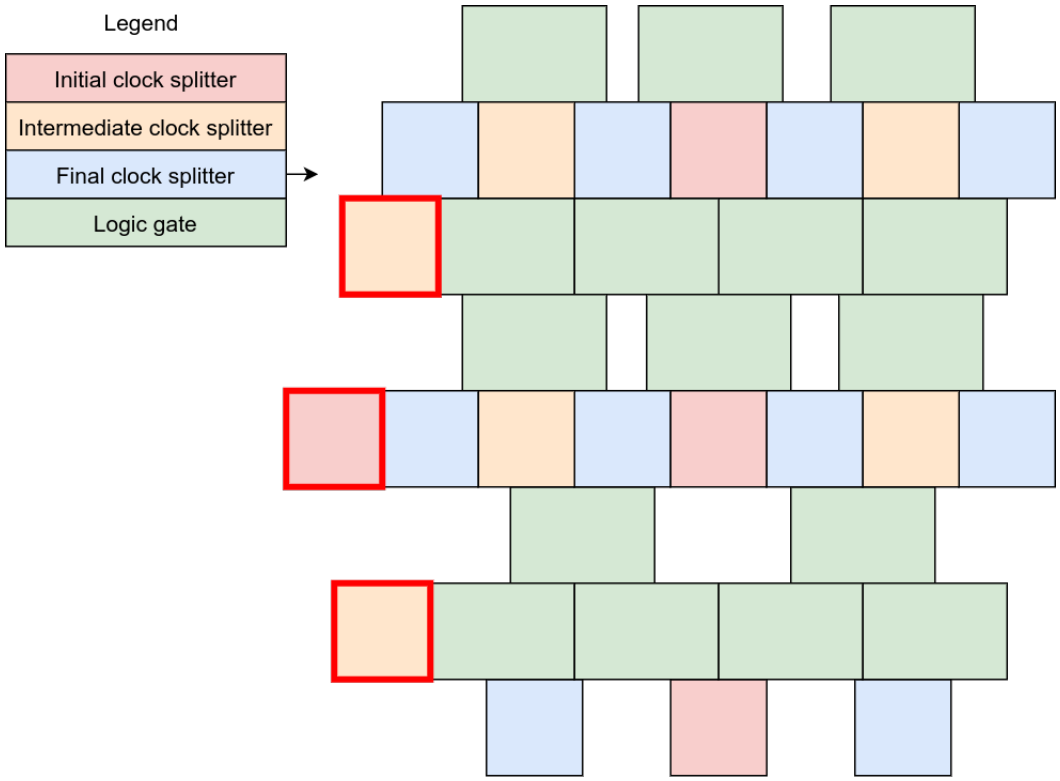

Figure 4.12: Main clock distribution network merged with the circuit

4.4.4. Cleaning Up

A cleanup of the clock network is done post-processing once all the clock splitter gates have been inserted and balanced. The objective is to remove or replace splitter gates that are redundant and do not contribute to the balancing of the clock network. The process iterates through all the clock gates and checks if the gate has only a single output connected if so it replaces the clock splitter with a buffer gate. The buffer gate has a single input and output that has the same time delay as a clock splitter. By replacing the excess clock splitters, it improves on power consumption due to the buffer gate consisting of less Josephson junctions and is smaller.

4.5. Cell Placement

With the layout finalised in Section 4.3 and the synthesis of the clock distribution network in Section 4.4 the only step left is to give the gates its final placement. Not many options are available of how to position circuits, however it still significantly affect the routing performance of the circuit. The layout being row-based, only allows for the gates to be shifted around horizontally. The gates are also designed to have a fixed height enabling the layout of the gates to be more uniform [35]. The horizontal placement schemes that were implemented are flush, centre and justify alignment.

The placement of all the gates must adhere to a $10\mu m \times 10\mu m$ grid, following current design guidelines [11]. To adhere to the regulation, the final position of all the gates are rounded off to $10\mu m$. The gates are placed from bottom to top, starting with the output pins and ending off with the input pins. The pins can be placed further away from the logic circuit and have wider spacing in between the pin if desired so.

4.5.1. Gate Attributes

The current method to describe the logic gates' physical attributes in a LEF file format is redundant and tedious. Therefore a custom file was implemented to allow for easy modification for any alterations to the logic gates. The custom file still represents the same data found in the LEF file but in a readable format. A TOML [36] file is utilised to describe the logic gates because of its simplicity. Listing 4.4 provides the template for how to describe the gates attributes. Only the bare essentials are required to describe the gate, namely the size, origin and pin locations relative to the bottom left corner of the gate. To help identify the direction of the pins, each pin is given a description namely, "in", "out", "clk" and "inout". Appendix E gives an extract of the logic gates developed for the Coldflux team.

```
[Gate_Template]
  size       = [x, y]      # size of gate
  origin     = [x, y]      # the origin of the gate
  pins       = ["", ""]    # the pin direction
  P1         = [, ]        # Pin 1
  Pn         = [, ]        # Pin n
  gate_delay = float       # picoseconds [ps]
```

Listing 4.4: Custom TOML file describing an AND gate

4.5.2. Flush Left and Right

Aligning the circuit's gates to the left or rightmost side is the simplest placement option. There is no need for calculating the spacing or aligning the gates due to the gates been stacked. The main drawbacks of stacking the gates to a side are that the interconnects have a higher deviation creating a greater difference in time delays with the interconnects. Another drawback is that having no space in between the gates creates less room for the router to place the interconnects, which may make the connections more complicated than

necessary and can lead to more noise. Figure 4.13 illustrates a section of a full adder circuit that is aligned flush to the left side and large open area is unutilised on the right side due to the difference is row widths. The complete placement of the full adder is in Appendix F.1.

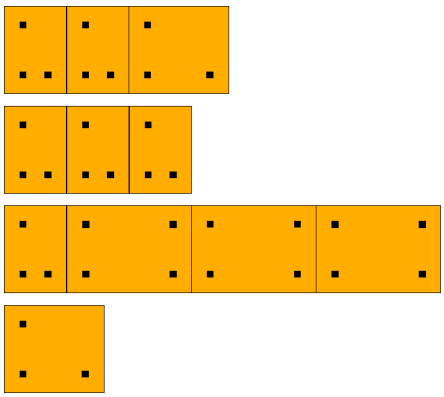

Figure 4.13: Flushed left layout of a circuit

4.5.3. Centring

Aligning the circuit's gates to the middle of the row is a more ideal placement option compared to flush alignment. The process starts by checking for the widest row to find the centre of the circuit, which is then used to adjust the starting point all the rows. The only drawback by centring the rows is that the lack of spacing in between the gates that can increase the interconnects congestion. By centring the gates, the standard deviations of the lengths of interconnects between gates are improved which potentially allows for an increase in clock speed of the circuit. Figure 4.14 portrays a segment of a full adder circuit with all the gates centred with no horizontal spacing, Appendix F.2 provides the complete circuit.

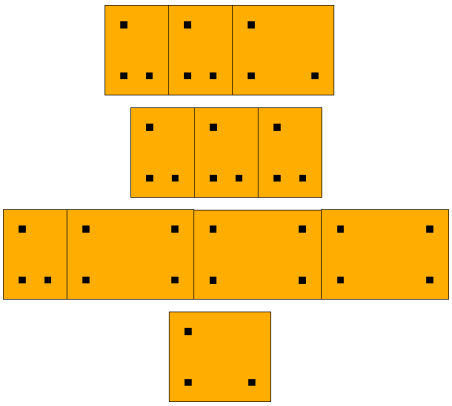

Figure 4.14: Centre aligned layout of a circuit

4.5.4. Centred and Full Justify

Justify alignment spaces each row's gates equally between each other with the widest row setting the edge limits. There are two types of justifying alignment, namely centred and full justify. The former spaces the gates equally from the edge as between the gates and the latter has the gates flushed on the row's edges. Full justify alignment has bigger gaps in between the gate increasing the standard deviation of the interconnect lengths between gates whereas with the centre justified's spaces between the gates is smaller, proving a better balance. Figure 4.15 and 4.16 demonstrates a segment of a full adder circuit that is centre and full justify respectively, clearly showing the difference in space between the gates. Appendix F.3 and F.4 gives the complete layout of the full adder circuits.

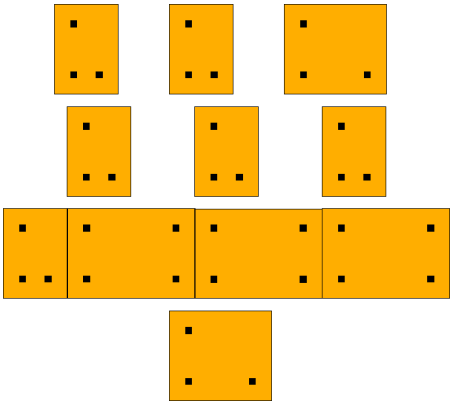

Figure 4.15: Centred justified layout of a circuit

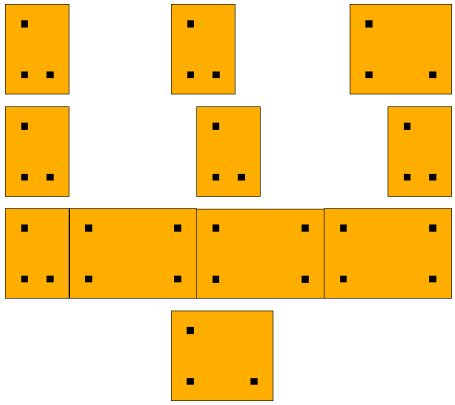

Figure 4.16: Full justified alignment of a circuit

Specific alignments of the gates may be better for different logic circuits, but generally for the larger-scale circuits, the more equally spaced the gates are the better performance. Therefore, centre justify alignment is the ideal choice for the horizontal alignment due to the best overall properties.

4.6. Routing Interconnects

The final step in synthesising a superconducting circuit is to route the interconnects(PTLs) between the gates. The tool of choice is Qrouter [25] due to it integrating well into the toolchain and its utilisation of LEF/DEF files. For Qrouter to operate a LEF file describing the physical attributes of the gates and DEF file containing the locations of the gates and interconnect paths is required as an input. A setup file is also required, but it remains mostly static unless the routing parameters and layer definitions need to be altered. Qrouters generates the results into the DEF file which then can be passed on to the next tool. Figure 4.17 demonstrates a visual example of the routing results that Qrouter generated.

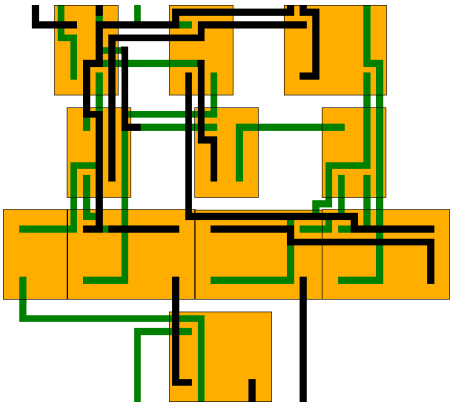

Figure 4.17: Routed interconnects(PTLs) of a circuit

The routing parameters are basic and is implemented as a cost function. The parameter which has the most significant effect on the performance of the routing is the cost of a via(PTL changing metal layers) per PTL. Therefore, the cost must be high compared to the rest of the parameters. Another significant parameter is the cost function to encourage the routing of the tracks to keep a specific direction(vertical or horizontal) on different layers.

A layout requirement of Qrouter is that the placement of the gates and pins must align to the grid otherwise, the tracks will fail to route or non-ideal routing may occur. The placement of the gates is already fixed to a grid in Section 4.5 but an offset is added ensuring it aligns to Qrouter's grid.

4.7. Summary

ViPeR consists of a set of tools that purpose is to automate the generation a complete superconducting circuit. ViPeR starts by interpreting an HDL(Verilog) description of a circuit into a gate-level representation. RSFQ features are then added to the gates enabling the circuit to operate under a superconducting state. Once electrical characteristics of the circuit are done, the optimal layout of the circuit is calculated followed by synthesising the clocking of the circuit. The final steps are to do the final placement of all the gates and then to route all the connections.

Only two parts of the toolchain require the use of external tools, namely ABC for circuit abstraction and Qrouter for interconnect routing. The tools have been repurposed from the CMOS industry and their feature set is consequently not fully applicable on the RSFQ field. Future releases may replace it with in-house tools that focus on making the tools more optimised for superconducting electronics. At the moment, the external tools are sufficient enough to be utilised.

The next steps before the final chip fabrication are to simulate the circuit to ensure that the circuit is working as intended using Die2Sim in Chapter 5. Once the circuit is electrically verified, then the final GDS file can be created of the circuit using ChipSmith discussed in Chapter 6 which implements all the filling requirements.

5 Circuit Verification - Die2Sim

5.1. Introduction

Electrical verification is a critical phase in the Verilog to chip fabrication progress, ensuring the success of the final circuit. The ideal method for the verification process is to run an electrical simulation on the circuit. The simulation will be able to show if the circuit is operating correctly and the clock speed at which the circuit can operate.

To get an electrical simulation running, the developed circuit which is described in an HDL format must be converted into a netlist along with the correct parameters for the simulation. The parameters must allow for different inputs and easy variation of clock speed to aid in the evaluation of the circuit. Several SPICE engines exist with support for Josephson junction namely JoSIM [37], WRspice [28], JSIM [29] and PSCAN [30]. JoSIM is the prefered choice due to performance and its been developed in-house for the Coldflux team.

This chapter will expand on the implementation and design choices of Die2Sim [38] along with the challenges to overcome the association issue with different naming schemes and the process for automating the testing. Die2Sim has been completely developed in C++ to ensure performance and the ability to integrate into other toolchains. A Python script was also created to aid in the evaluation of the PTLs.

5.2. Implementation

5.2.1. Design flow

The verification process starts by parsing the components and nets properties from the DEF file into memory. During the importing process, the components, gate netlists and nets are stored in a binary tree due to the large size of the circuits which will ensure that access to the data is swift. Most circuit design teams have different naming schemes for their circuits, causing an issue with matching up the gates' netlist, input and output pins. To overcome the different naming schemes, a translating table is utilised to ensure that the netlists are associated correctly. Section 5.2.2 expands on linking the different names of the components and pins. Figure 5.1 illustrates the flow of how Die2Sim creates a JoSIM

file which can then be simulated.

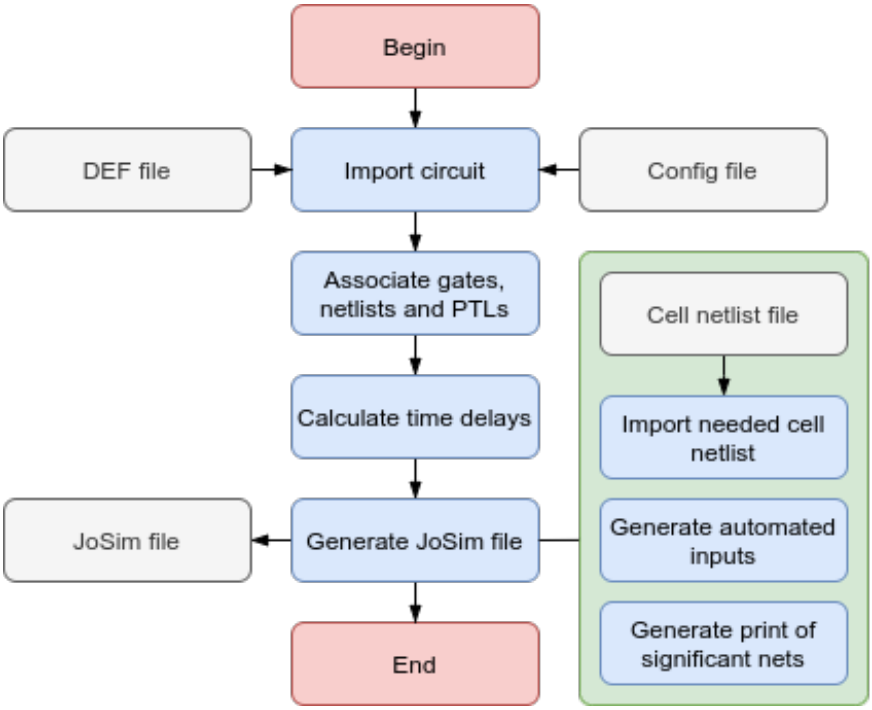

Figure 5.1: The flow of Die2Sim

Using the path coordinate of the interconnects from the DEF file, the length of the connections can be calculated. The lengths are then used to calculate the propagation time delays on the PTLs, the speed of the SFQ pulse is mentioned in Section 2.4. The time delay of the SFQ pulse through vias and around corners is insignificant to be calculated. PTL has other characteristics, such as noise and frequency response but it is currently ignored due to insufficient modelling.

Creating the JoSIM file starts by importing the netlists of the gates specified from a list in the config file. Then the PTLs with their time delay and the components which are linked to the gates' netlist in its own sub-circuit. Placing the circuit in a sub-circuit enables the user to incorporate the circuit into other designs if needed. Once the sub-circuit is generated, the clock and input signals are created each with their own current source, DCSFQ and PTL transmitter. Automating the input patterns and verifying the output results will be expanded in Section 5.4. To finalise the SPICE file, all the inputs and outputs of the circuit is printout to a CSV file.

5.2.2. Translation Table

Development of superconducting circuits requires multiple people if not teams to create a working circuit, this may cause issues to arise with a default naming scheme across the board. The solution is a translating table that interprets the circuit's(DEF file) gates and

then to link it to the correct netlists. The tables are defined in the config file using the TOML [36] language.

Another naming scheme drawback is the identifiers indicating what pins are inputs, outputs and clock signals. The identifiers generally are several characters, for example "SUM" or "O" may indicate an output pin. To identify the pins, a table of possible input, output and clock pin identifiers is implemented. A search is then conducted through the pins looking if the identifier appears anywhere in the name and then is allocated to the correct pin type.

5.3. PTL Statistics

An essential part of developing circuits is to evaluate the routing and placements of the gates. Analysing the PTL time delay distribution provides a quick way to determine what placement and routing methods are more efficient. Outlier PTLs which can cause timing issues can also be easily identified. To analyse the circuit's routing, a Python script using Jupyter [39] allows for easy visualisation of the data.

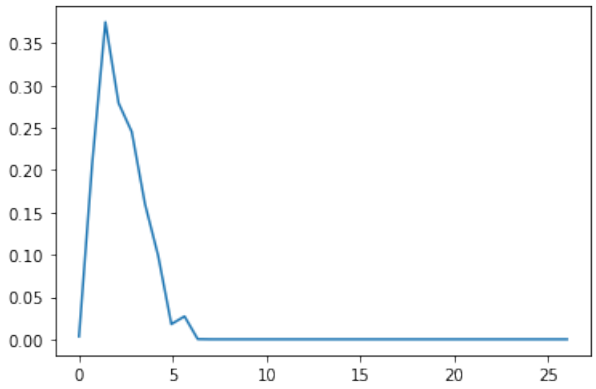

Figure 5.2: PTL time delay distribution of a full adder

```
std: 1.14
min: 0.7
max: 5.4
mean: 2.24
```

Listing 5.1: PTL statistics for a full adder circuit

The Python script displays the distribution of the PTLs as depicted in Figure 5.2 and can provide common statistics as illustrated in Listing 5.1. The ideal PTL statistics must have a low standard deviation and the maximum and minimum time delays should be close as possible together ensuring that the SFQ pulses between gates have the same time of arrival. Certain tracks as with the clock tree interconnects are allowed to be longer because it has minimal influence on performance due to it being balanced.

5.4. Automated Verification

The goal of automated testing is to verify that a circuit is working with minimal input from the user. It falls into two parts, adding the necessary components to the netlist so that the circuit can be simulated and for the tools to automatically generate input test patterns and check if the expected results are achieved.

5.4.1. Test Bench

The first step in the verification process is to connect the supporting components to the circuit input and output pins. The input pins require a DCSFQ gate followed by a passive transmission lines transmitter (PTLTX) with the input signal being produced by a current source. The output pins are connected to a passive transmission lines receiver (PTLRX) in series with a resistor to ground. The output signal is the measured voltage or phase across the resistor.

5.4.2. Input Pattern Generation and Result Analysis

Future development of Die2Sim is to generate its own input test patterns and test if the results are correct. Multiple simulations may have to be conducted to ensure that all the test cases work as per the circuit's description. The tool will have to provide different input patterns at different timing intervals to obverse the circuits operating margins. Analysis of the output pin results must be checked for the correct signals and unexpected pulses. Another significant test will examine the circuit for different clock speeds to determine the maximum clock speed at which the circuit can operate.

5.5. Summary

Verifying the circuit is a crucial stage in delivering a circuit to production. The purpose of Die2Sim is to rapidly generate test benches for developed circuits. Die2Sim can import a circuit and convert it to a JoSIM SPICE file in a test bench. The circuit is then simulated and results can then be verified. Different circuit layout methods can be evaluated by analysing the PTL time delays using the accompanying Python script. After the verification process is done, the final layout of the circuit can be generated using chipSmith which is examined in Chapter 6.

6 Chip Synthesis - chipSmith

6.1. Introduction

chipSmith handles the final synthesis procedure before chip fabrication can transpire. The primary objective is to import all the gates designs(GDS files) and place them per the designed layout from Chapter 4. Additional requirements must also be met, i.e. the fill around the structures(gates and tracks) and the distribution of the DC bias lines. A custom library was developed in order to handle all the operations needed to generate a Graphic Design System (GDSII) file.

This chapter look at the implementation and dependencies of chipSmith. A quick overview of have placement of gates and routing will be expanded then followed by the biasing of the gates. Once the chip is populated, the fill can be placed around all the structures. Finally, the custom GDS library will be discussed with all its features that aid the generation of the final chip design.

6.2. Placement and Routing

The placement and routing is the first stage in the chip synthesis process. Figure 6.1 illustrates the end product of the placement and routing. The process starts by importing the routed DEF file and the GDS files of the gates. To be memory efficient for large scale circuits, when chipSmith populates the chip, only the references of the gate are used preventing redundancies in the GDS file. Routing of the PTLs is straight forward due to the GDS format that supports interconnects(paths).

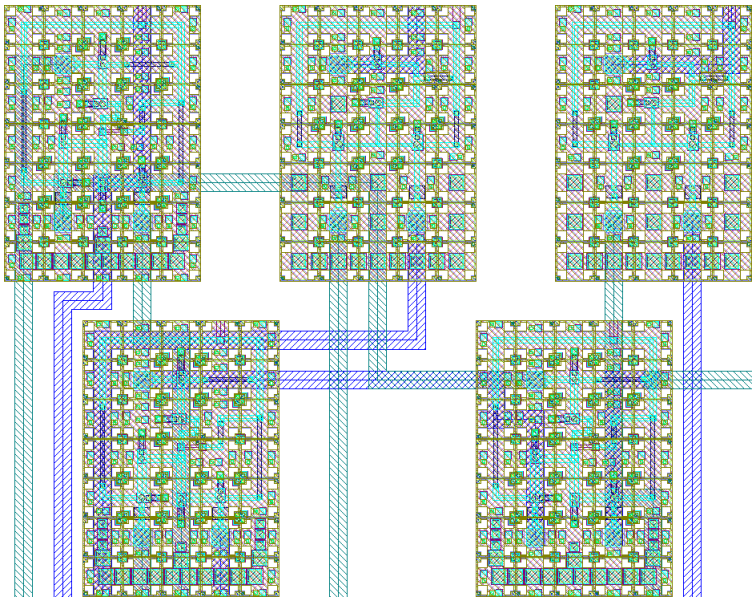

Figure 6.1: Extract of a full adder's final placement and routing

6.3. Biasing

In addition to the signal interconnects, the gate requires a separate input line which provides the DC biasing current. For RSFQ gates to operate they require a DC current in order to bias the Josephson junctions. Due to the layout of the gates being row-based and having the DC bias input at the top edge, it allows for straight forward routing. The gates are positioned to have a vertical gap in between them to allow for the DC bias line then a short trace connects the gate to the DC biasing line. Figure 6.2 provides an example of how the gates are placed so that it can be biased.

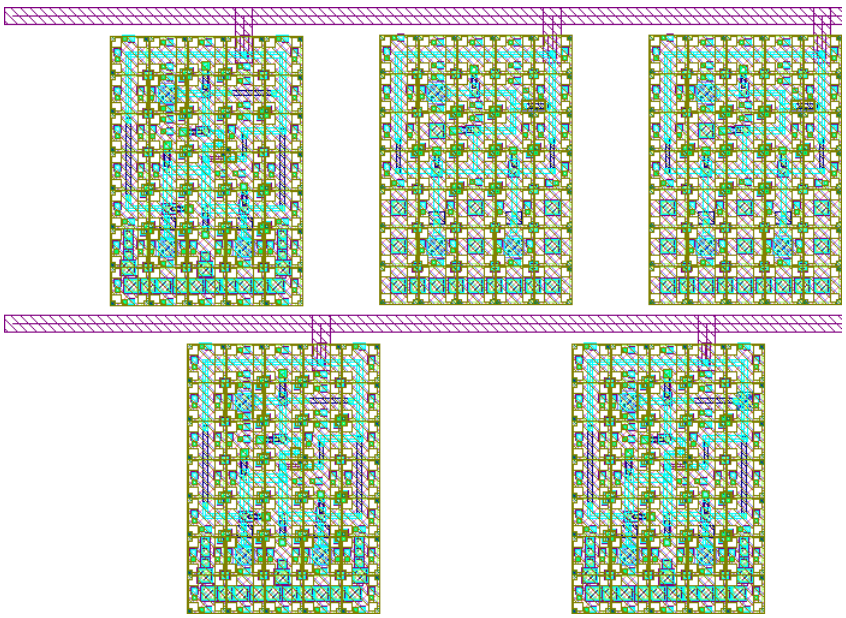

Figure 6.2: Biasing lines of a full adder circuit

Unlike the standard RSFQ signal interconnects which can only have one input and one output termination. The biasing line can have multiple connections provided that the total biasing current does not exceed the conductors critical current. Future development will calculate each row's critical current to determine which biasing rows can be connected, provided the track is still under critical current.

6.4. Fill

Fill has a significant influence on the performance of the final circuit. It ensures that it adheres to the design rule checking (DRC) density check, shielding the circuitry and creates motes helping to diverge external magnetic fields. Each metal layer has a different fill pattern which must be filled around all the structures as demonstrated in Figure 6.3 and 6.4 with the fill around the different routing layers. [11]

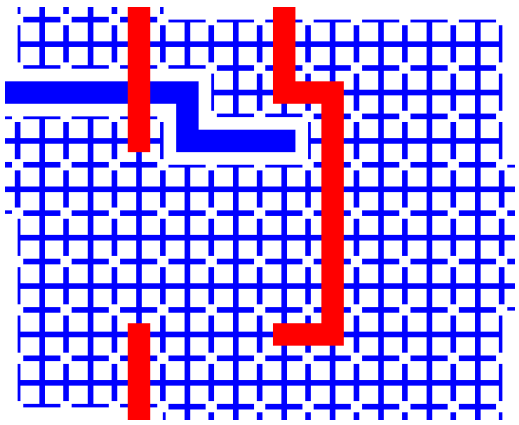

Figure 6.3: Fill around metal layer 1

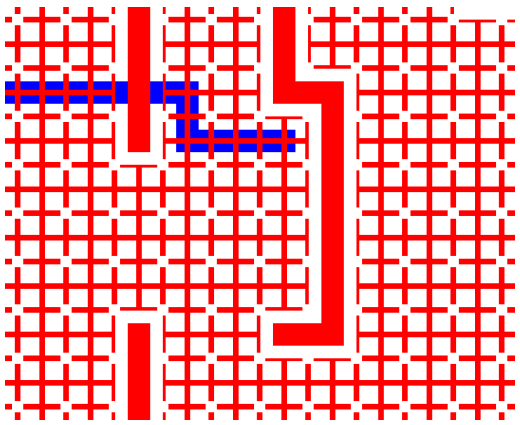

Figure 6.4: Fill around metal layer 3

The process starts by importing the GDS files of the fill patterns as with the gates in Section 6.2. A matrix is generated representing the fill grid of the different metal layers. With all the gates outlines forming a rectangle and the tracks follow Manhattan routing, a rectangle or subsets of rectangles can represent all the structures. To determine what coordinates on the grid needs fill or not, all the rectangles structures are iterated through. The coordinates(points) within the rectangles are taken and set in the matrix not to have fill. The matrix is then iterated through placing the fill in the final GDS file.

6.5. GDScpp

To aid the development of chipSmith, a GDS C++ library was created called GDScpp [40]. The library fully supports the GDS format enabling it to read and write GDS files. The main objective was to create a feature set that can aid the rapid generation of the RSFQ circuits.

6.5.1. Importing Files

The main feature required is the ability to import multiple GDS files into a single file. Two methods were implemented, a direct copy and then the ability to completely copy the file into memory.

The direct copy method merges multiple GDS files directly into another single file. It was initially created when chipSmith only did the placement of the gates. The benefit is that the data that is copied is an exact match to the original with zero alterations.

Copying the GDS file into memory was developed later when the need for manipulating the geometry was required. For the GDS file to be imported into memory, the complete GDS standard had to be supported. By copying the file into memory and then outputting it to a new file, it can create minor deviations in the file structures, i.e. the order of the data but the data left unaltered.

When importing multiple files especially of gates, a high chance exists that the files may contain similar structures. In a GDS file all the structures must have a unique identifier, therefore redundancy checking is required. Every time a file gets copied or imported into memory, it must get checked if it does not already exist. If it already exists, it is ignored. When coping multiple gates(GDS files) and inserting them into a single file, often the final file is smaller compared to the sum of the gate because of the removal of redundancies.

6.5.2. Generation

The ability to generate a GDS file proved to be a crucial step in order to create the final circuit placement. Generating a GDS file was broken up into three levels, namely low, middle and high level. It is designed so that the user only needs to interface with the highest level providing a form of error checking. The low level is designed to handle all the binary operations seeing that the GDS format uses its own method for storing data. Intermediate level manages all the generation of the different GDS records along with functions to improve the handling of the geometry. The final and highest level creates the complete GDS file, including all the overheads with minimal input from the user.

6.6. Summary

The whole chip layout process can be long and tedious, especially for large scale designs which can cause the designer to make mistakes. This stemmed the need for the development of chipSmith to automate the synthesis of the final chip in GDS format. This chapter gave an overview of the generation of the gate and PTLs, the biasing of the gates, filling the void area and the library implemented to aid the generation of the final GDS file.

For future development, the goal is to generate a complete chip with minimal input from the designer. Connecting all the signal and biasing lines to the chip's pads and allowing for more complex fill shapes will enable diagonal tracks and non-rectangular structures.

7 Results

7.1. Introduction

This chapter will analyse the performance of the circuit synthesis tool(ViPeR) describe in Chapter 4. In order to analyse the circuits, the circuit verification tool (Die2Sim) from Chapter 5 is utilised to aid the testing. Several different combinational circuits are used to show how the circuit synthesis handles different types of circuits.

The first circuit that is synthesised is a full adder circuit that is considered small scale. A quick overview is given of simulation to check the expected results with different input patterns. The second analysis is a 4-bit Kogge–Stone adder (KSA) circuit being significantly larger than the full adder. The 4-bit KSA is big enough to stress test the synthesis algorithms, ensuring that the methods used are scalable. An in-depth analysis is given of the circuit and a comparison to a pre-existing KSA. The final analysis is of an 8-bit KSA circuit as it pushes the limits on the size of circuits that the synthesis tool can handle.

7.2. Small Circuits

A fuller adder circuit is relatively small with only 23 gates and 38 PTLs. With the circuit being small, the placement and routing of the gates are straight forward due to the limited layout options. Appendix G provides the final layout of the full adder.

An electrical simulation using JoSIM was utilised to generate Table 7.1 consisting of all the possible input patterns and expected results. The results are as expected showing that the circuit logic is working accordingly. Figure 7.1 and 7.2 demonstrates two of the simulations that were executed and the results can clearly be seen.

A_IN	B_IN	C_CIN	SUM	C_OUT
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 7.1: Full adder truth table

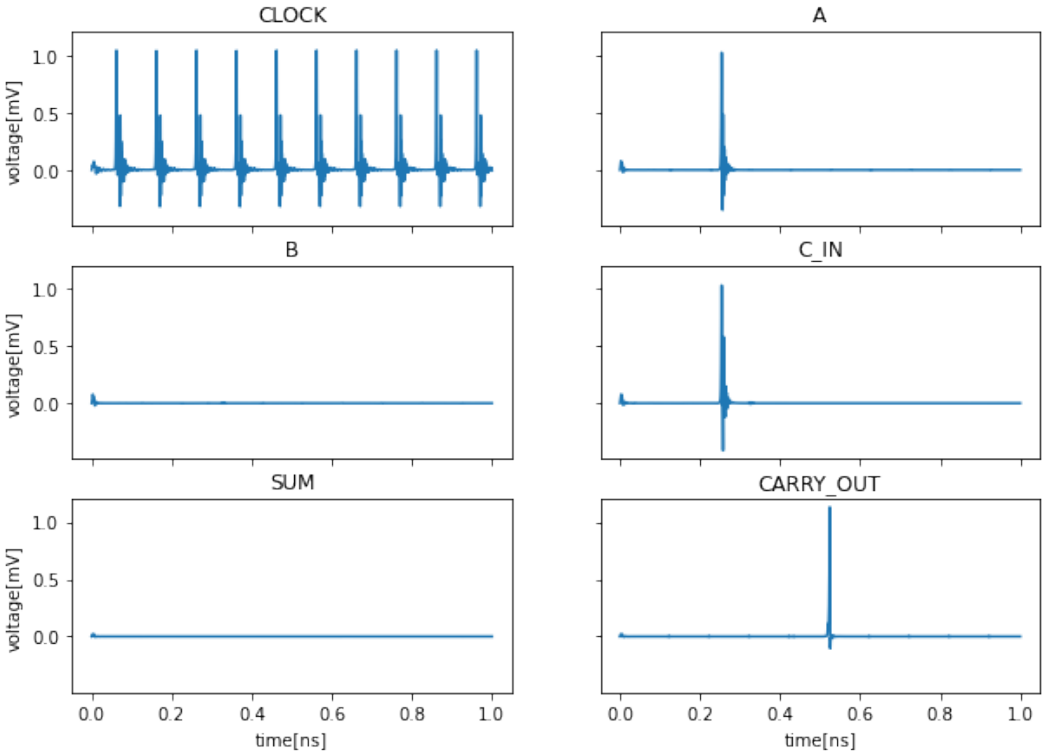

Figure 7.1: JoSIM simulation of full adder

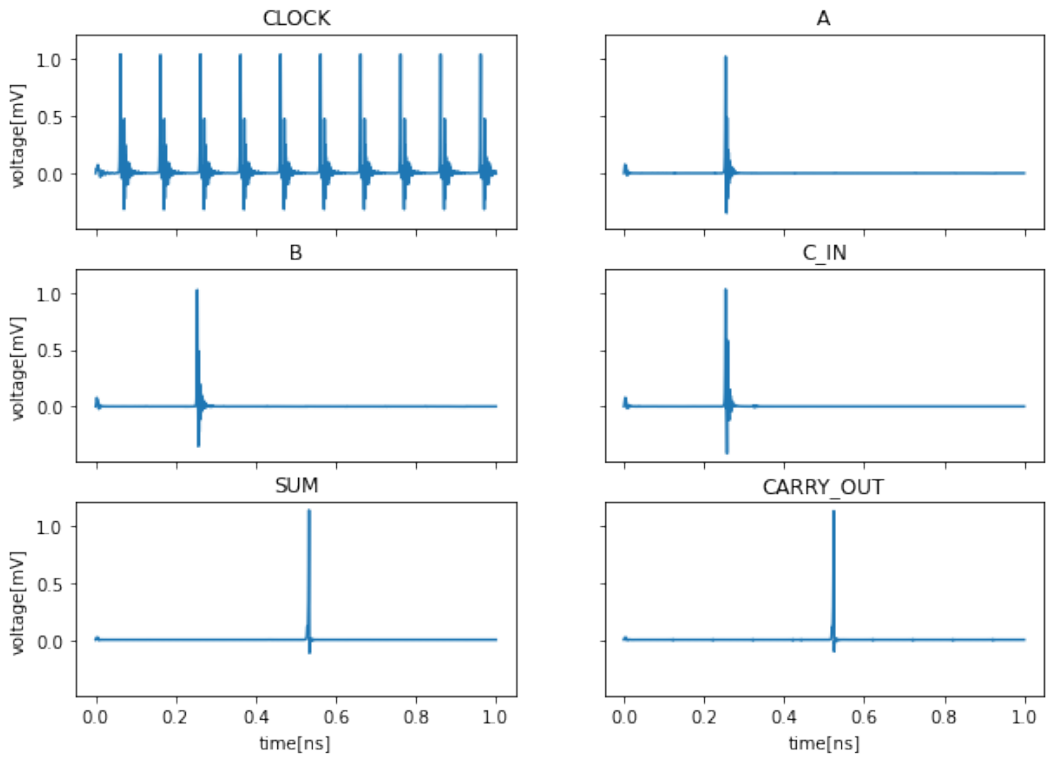

Figure 7.2: JoSIM simulation of full adder

7.3. Mid Scale Circuits

The 4-bit KSA circuit is all-rounder circuit that is used as a benchmark with other tools. The synthesising the KSA circuit using ViPeR takes only 1 second to execute and takes Qrouter 0.5 seconds to route all the nets. The whole process takes a total of 1.5 seconds to run, looking instantaneous to the user.

7.3.1. Simulation

Electrical simulating a circuit is the ideal method to verify if a circuit is operating correctly. The simulation provides insight if the logic of the circuit is operating as expected by observing the results, ensuring that the timing of the circuit is sufficient and to test the operating limits, for example the clock speed.

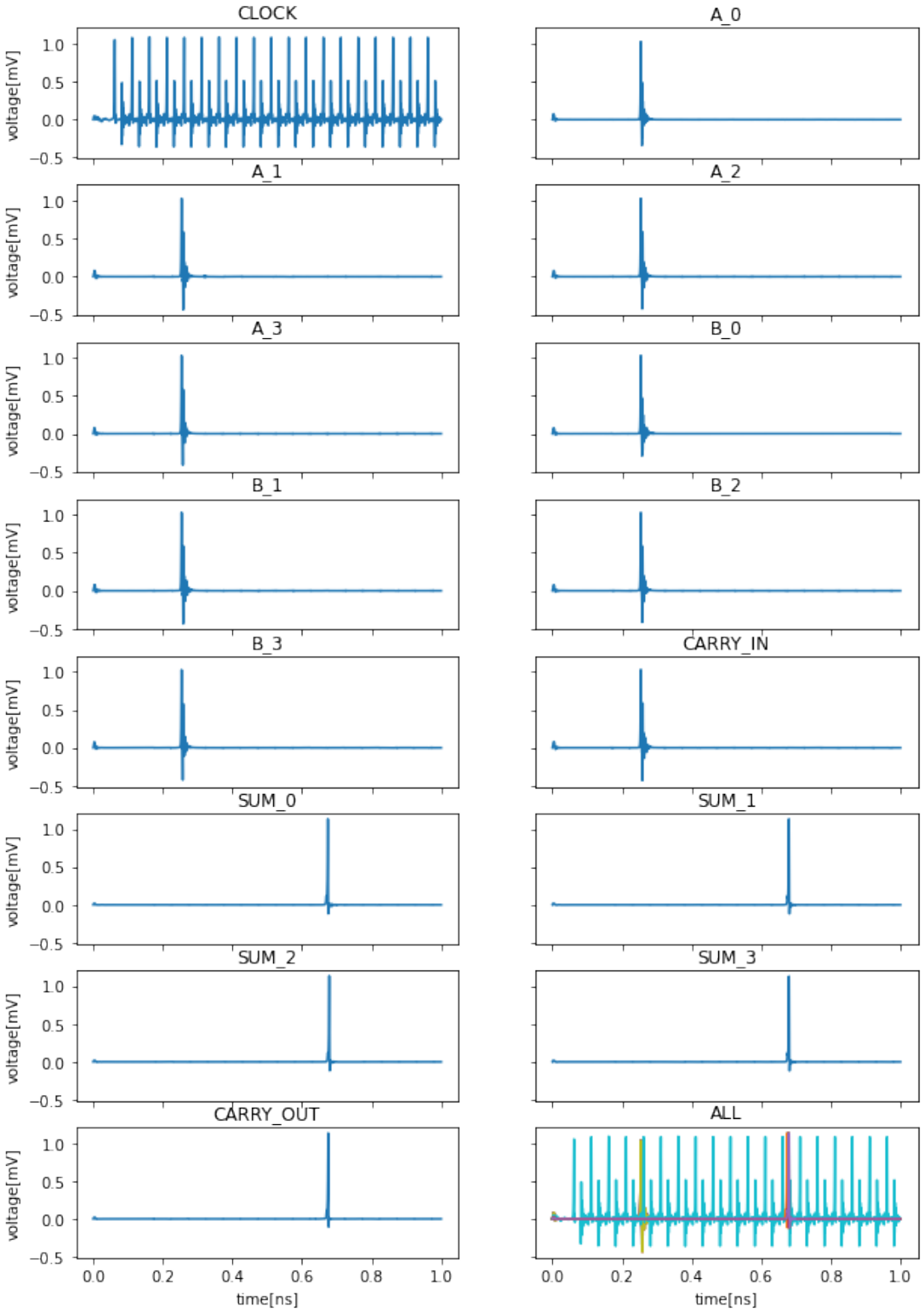

Figure 7.3: JoSIM simulation of 4 bit KSA

Figure 7.3 provides measurements of all the 4-bit KSA pins. The inputs pins(A_n; B_n; CARRY_IN, CLOCK) can be seen inserting the input signal and the output pins(SUM_n; CARRY_OUT) producing the results. The output signals are all produced at the exact time as illustrated in Figure 7.4 showing that the circuit is well balanced.

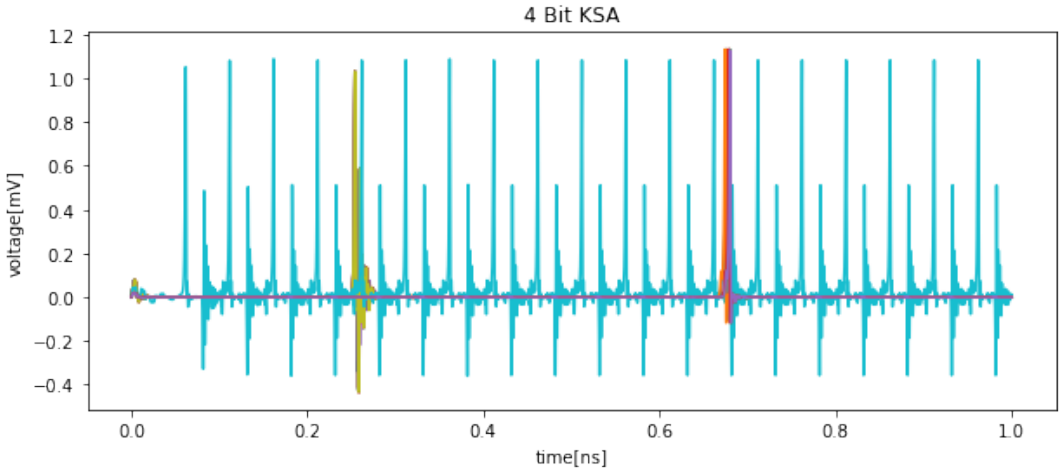

Figure 7.4: JoSIM simulation of 4 bit KSA

7.3.2. Placement and Routing

The placement and routing of the KSA proved to be successful in Section 7.3.1. The final GDS placement with the pin labels are illustrated in Figure 7.6. Routing of the PTLs is one of the most significant factors on the performance of the circuit and placement of the gates influences it. The maximum clock speed of the circuit is $24GHz$. If the PTL delays are fixed to $1ps$, the maximum clock speed is $34GHz$, proving that there is room for improvement with the layout of the gates.

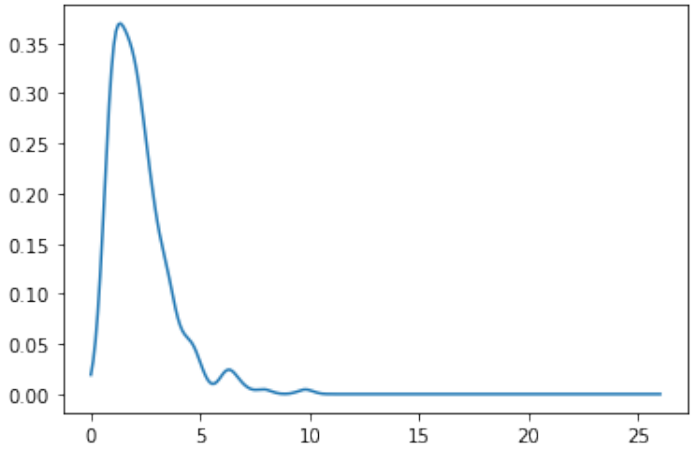

Figure 7.5: PTL delay distribution of KSA 4 bit circuit

The ideal routing is to have a small deviation of the PTL delays in order to keep the delays constant across the circuit. The maximum time delay of a PTL limits the clock speed at which the circuit can run, therefore it is necessary to ensure it stays small. Listing 7.1 provides the PTL's statistics of the 4-bit KSA and Figure 7.5 illustrates the distribution on the PTL delays. The low standard deviation and average delay provide ideal characteristics for the circuit to operate in, however the maximum delay is high.

The higher maximum PTL delay is caused by the initial PTL that feeds into the H-tree clock. The distance between the clock input pin and the first splitter(branch) is long due to the input pin being on the edge of the circuit and the initial splitter is placed in the centre. Due to the clock distribution network being balanced, the higher delay primarily only affects the circuit's setup uptime.

```
standard deviation: 1.39
minimum delay[ps]:  0.4
maximum delay[ps]:  9.8
mean delay[ps]:     2.23
```

Listing 7.1: PTL statistics of a 4 bit KSA4

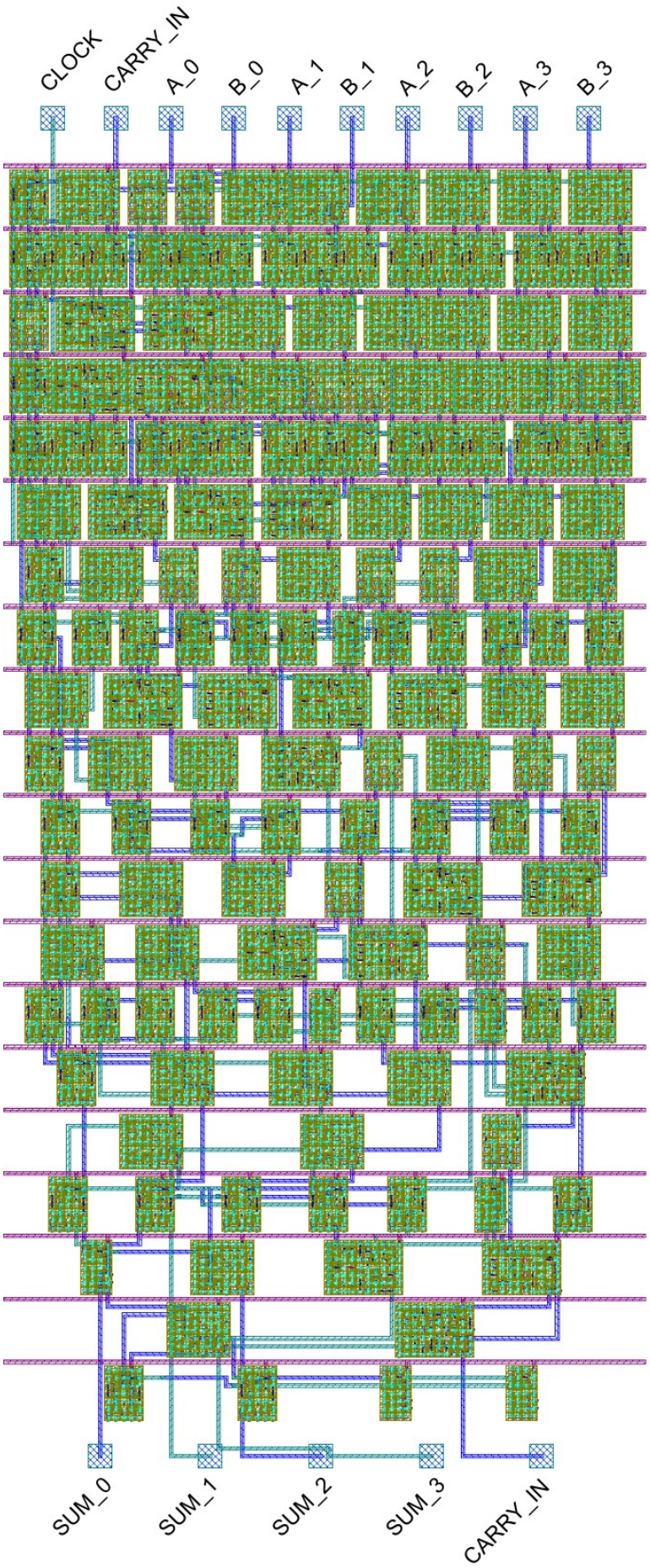

Figure 7.6: Final GDS layout of a 4-bit KSA

7.3.3. Comparasion with USC's qPALACE

Comparing USC qPALACE's 4-bit KSA circuit with Stellenbosch University's ViPeR, only marginal differences can be observed. Table 7.2 gives a summary of the differences between the two tools.

	qPALACE	ViPeR	Difference
Clock speed[GHz]	**26**	24	2
Operation time[ns]	**0.27**	0.38	0.11
Gate count	**158**	160	2
PTL count	260	**250**	10

Table 7.2: Comparison of a 4 bit KSA from qPALACE and ViPeR

The first difference is the maximum clock speed at which the circuits operate. ViPeR's KSA can operate at a maximum of $24GHz$ while qPALACE's achieves a speed of $26GHz$, the difference been $2GHz$. The cause for the difference may be that qPALACE has a more complex placement algorithm for only an 8.3% gain over ViPeR.

The second difference is the duration of an operation, the time it takes from the input signal being inserted into the circuit until the result is produced. The circuit were both tested at $24GHz$ and the input signal was injected at $0.25ns$. The difference of $0.11ns$ equates to 2 clock cycles, showing that the difference originates from the logic synthesis of the circuits. qPALACE's algorithms optimise the circuit to have fewer clock levels allowing the circuit to produce the results faster.

The final comparison is the number of gates and nets(PTLs/interconnects) in the circuit excluding the pads. qPALACE's KSA has a total of 158 gates and 260 nets while ViPeR has 160 gates and 250 nets. The difference is not significant, however having more nets can cause the router to run for longer.

7.4. Large Scale Circuits

To demonstrate the capabilities of the circuit synthesis algorithms, an 8-bit KSA was synthesised. Its has 539 gates and 823 nets(PTLs) making it over three times larger than the 4-bit KSA. The increase in size helps to verify the feasibility of the synthesis algorithms. The 8-bit KSA was successfully synthesised with a minor timing error.

With larger size circuits, it emerged that a splitter with a higher fanout was required. To increase the fanout, a simple nested(tree) structure of splitter is utilised, Figure 7.7 demonstrates. The splitters do have unconnected outputs, but it ensures that

the tree remains balanced. The major drawback of the splitter tree is that it increases the propagation delay of the SFQ pulse between the gates decreasing the maximum clock speed of the circuit.

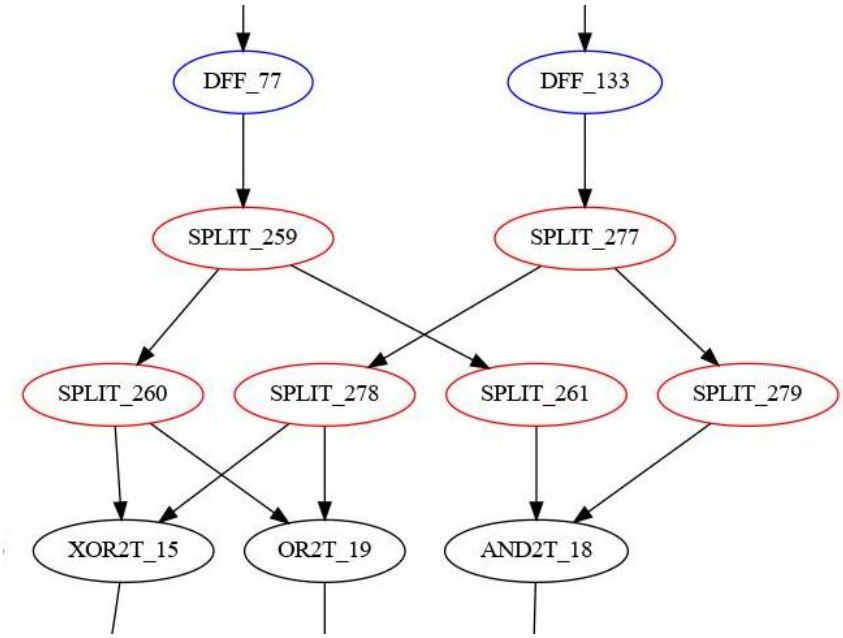

Figure 7.7: Logic flow of a splitter tree in an 8 bit KSA circuit

```
standard deviation: 2.11
minimum delay[ps]:  0.4
maximum delay[ps]:  17.4
mean delay[ps]:     2.64
```

Listing 7.2: PTL statistics of a 8 bit KSA4

The timing issue can be easily seen in Figure 7.8 in the CARRY_OUT graph(pin). The double pulse indicates a timing violation in the CARRY_OUT signal branch caused by excessively long PTLs. The longer PTLs are caused by the logic gates being further apart due to the optimal layout algorithm not handling the large layout correctly. Listing 7.2 illustrates that the circuit has a high deviation of PTL time delays indicating that the timing issue is prevalent. If the PTL delays are all fixed to the same time delay, the timing violation disappears, proving that the gates' placement and routing are not optimal for larger-scale circuits.

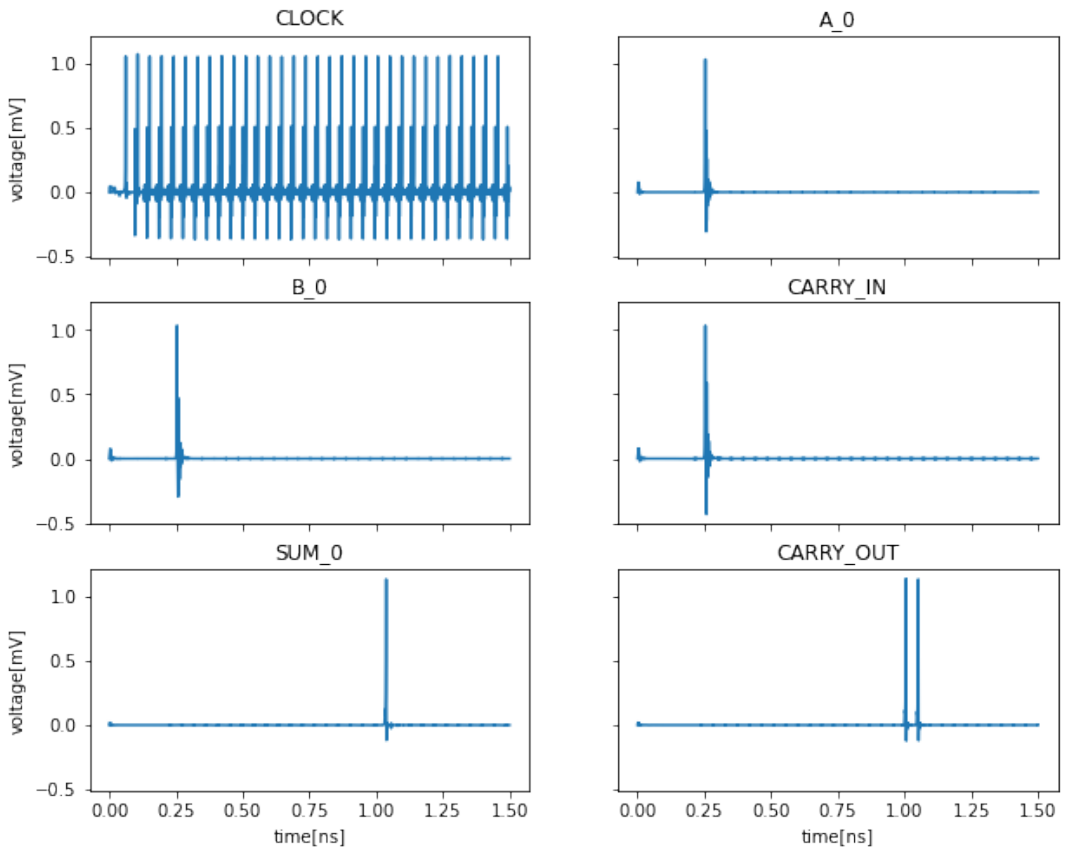

Figure 7.8: JoSIM simulation of 8 bit KSA

7.5. Summary

In this chapter, the synthesising of circuits using ViPeR was analysed with different circuits. Die2Sim was used throughout the testing to generate the SPICE(JoSIM) files used to simulate the circuits. All the test results are favourable with room for some improvements.

All the tests consisted of different combinational logic circuits trying to show how the chip synthesis fairs under different situations. The first circuit was a full adder circuit demonstrating that the circuit's logic is operating accordingly. The second circuit consisted of an analysis of a 4-bit KSA and was compared with a pre-existing KSA circuit showing that the results are favourable. The final analysis was of an 8-bit KSA which started to show where improvements are required to be made for large scale circuits.

8 Conclusion and Recommendations

8.1. Conclusion

The dissertation examines the procedure required to automate the synthesis process of a superconducting logic circuit using RSFQ technology. A curated set of tools was created to handle all the different stages of the circuit design process, namely circuit synthesis(ViPeR), verification(Die2Sim) and final chip design(chipSmith).

The synthesis process begins in ViPeR, where a high-level description of a logic circuit is generated into a complete RSFQ chip. During the process the circuit gets described at gate level, is balanced and the fanout out of the gates is corrected. The gates are then individually clocked with a balanced clock distribution network. Then all the gates are generated, the optimal layout is calculated allowing for the final placement and routing of the gates.

With the circuit synthesis complete, the verification process using Die2Sim and JoSIM(SPICE engine) puts the circuit through its paces. Different input patterns are fed into the circuit and checked if the results are as expected. The maximum clock speed at which the circuit can operate at is also tested, indicating the performance. If any issues arise, the problem is easily debugged by stepping through the simulation and observing where any discrepancies may be.

The whole process is completed with the final placement synthesis of the circuit using chipSmith to generate the GDS file. The placed and routed circuit is taken from ViPeR and the RSFQ gates layout is linked together into a single GDS file. chipSmith also connects the DC bias lines and fill is placed throughout the circuit to ensure that the chip adheres to the DRC.

The toolchain was evaluated by synthesising several circuits and examining the final simulation results. The small and mid-scale circuits were synthesised and simulated successfully. The mid-size circuit being a 4-bit KSA, was compared to a pre-existing one and the performance of the two circuits are almost identical proving that the circuit synthesis tool, ViPeR is on par with international efforts. An 8-bit KSA circuit representing a large scale circuit was also synthesised with only a minor timing violation

occurring, showing that the current layout algorithm needs more attention in order to allow for synthesis of bigger circuits.

The toolchain achieved its goal, to synthesise a complete RSFQ circuit swiftly. The synthesised 4-bit KSA circuit that ViPeR generated operates near flawlessly when compared qPALACE's KSA. The current work lays the foundation for future work to facilitate the synthesis of significantly larger circuits and improve operating speeds.

8.2. Future Improvements

Synthesising superconducting circuits consists of many intricate parts which all has a significant influence on the final product. Improvements can be made throughout the whole toolchain to increase performance and synthesise execution time. The parts that most likely need improvements are the logic synthesis and optimal layout algorithm.

The current logic synthesis only allows for combinational logic limiting the complexity of the types of circuits that can be created. Sequential(feedback) logic is often used in circuitry that requires some form of memory and the main issue arises when trying to balance the circuit with DFFs. Therefore, an improved balancing and optimal layout algorithm will have to be implemented to handle sequential logic.

Being able to optimally layout massive(larger than large) scale circuits is a vital requirement for designing useful logic circuits for modern-day application. For the adoption of RSFQ CPU to become applicable in industry, the CPU must be able to handle 32-bit if not 64-bit logic. By looking at the size difference between 4 and 8-bit KSA circuit, a 32 or 64-bit logic circuit will be in the order of magnitude larger. In order to achieve massive-scale logic circuits, an improved or new optimal layout algorithm will be needed to handle the placement and routing of all the gates effectively.

A further layout improvement can be to place the splitter gate next to its input gate, resulting in a significant improvement to the overall layout. It will ensure that the rows are clock level-based, which can dramatically improve timing performance and improve the variance of the interconnect lengths towards the output(bottom) of the circuit.

Bibliography

[1] D. S. Holmes and B. A. Hamilton, "Superconducting Hybrid Systems," *IEEE Comput. Soc.*, 2015.

[2] G. I. Seffers, "If You Build the Tools, Superconducting Electronics Will Come," 2016. [Online]. Available: https://www.afcea.org/content/?q=Article-if-you-build-tools-superconducting-electronics-will-come

[3] D. C. Brock, "Will the NSA Finally Build Its Superconducting Spy Computer?" 2016. [Online]. Available: https://spectrum.ieee.org/tech-history/silicon-revolution/will-the-nsa-finally-build-its-superconducting-spy-computer

[4] K. K. Likharev and V. K. Semenov, "RSFQ Logic/Memory Family: A New Josephson-Junction Technology for Sub-Terahertz-Clock-Frequency," *IEEE Transactions on Applied Superconductivity*, vol. 1, no. 1, 1991.

[5] W. Chen, A. V. Rylyakov, V. Patel, J. E. Lukens, and K. K. Likharev, "Rapid single flux quantum t-flip flop operating up to 770 GHz," *IEEE Trans. Appl. Supercond.*, vol. 9, no. 2 PART 3, pp. 3212–3215, 1999.

[6] R. Newrock, "What are Josephson Junctions? How do they work?" p. 1, 1997. [Online]. Available: https://www.scientificamerican.com/article/what-are-josephson-juncti/

[7] S. K. Tolpygo, V. Bolkhovsky, T. J. Weir, A. Wynn, D. E. Oates, L. M. Johnson, and M. A. Gouker, "Advanced fabrication processes for superconducting very large-scale integrated circuits," *IEEE Trans. Appl. Supercond.*, vol. 26, no. 3, pp. 1–10, 2016.

[8] H. Herbst, "Gate-Level Superconductor Integrated Circuit Fabrication Process Modelling for Improved Layout Extraction," 2020.

[9] C. Fourie, "Single flux quantum circuit technology and CAD overview," *IEEE/ACM Int. Conf. Comput. Des. Dig. Tech. Pap. ICCAD*, pp. 15–20, 2018.

[10] B. Dimov, V. Todorov, V. Mladenov, and F. H. Uhlmann, "Improved techniques for long-distance signal propagation within the Rapid Single-Flux Quantum digital circuits," *ISSCS 2005 Int. Symp. Signals, Circuits Syst. - Proc.*, vol. 2, pp. 733–736, 2005.

[11] C. J. Fourie, C. L. Ayala, L. Schindler, T. Tanaka, and N. Yoshikawa, "Design and Characterization of Track Routing Architecture for RSFQ and AQFP Circuits in a Multilayer Process," *IEEE Transactions on Applied Superconductivity*, vol. 30, no. 6, pp. 1–9, apr 2020.

[12] S. N. Shahsavani, T. R. Lin, A. Shafaei, C. J. Fourie, and M. Pedram, "An integrated row-based cell placement and interconnect synthesis tool for large sfq logic circuits," *IEEE Trans. Appl. Supercond.*, vol. 27, no. 4, 2017.

[13] K. Gaj, E. G. Friedman, M. J. Feldman, and A. Krasniewski, "A Clock Distribution Scheme for Large RSFQ Circuits," *IEEE Trans. Appl. Supercond.*, vol. 5, no. 2, pp. 3320–3324, 1995.

[14] C. A. Mancini, N. Vukovic, A. M. Herrf, K. Gaj, M. I. Bocko, and M. J. Feldman, "RSFQ circular shift registers," *IEEE Trans. Appl. Supercond.*, vol. 7, no. 2 PART 3, pp. 2832–2835, 1997.

[15] R. N. Tadros and P. A. Beerel, "A Robust and Tree-Free Hybrid Clocking Technique for RSFQ Circuits - CSR Application," *2017 16th Int. Supercond. Electron. Conf. ISEC 2017*, vol. 2018-January, pp. 1–4, 2018.

[16] K. Jackman and C. J. Fourie, "Flux trapping experiments to verify simulation models," *Supercond. Sci. Technol.*, vol. 33, no. 10, p. 105001, 2020.

[17] V. K. Semenov and M. M. Khapaev, "How Moats Protect Superconductor Films From Flux Trapping," *IEEE Trans. Appl. Supercond.*, vol. 26, no. 3, 2016.

[18] C. J. Fourie, "Digital Superconducting Electronics Design Tools-Status and Roadmap," *IEEE Transactions on Applied Superconductivity*, vol. 28, no. 5, 2018.

[19] R. S. Bakolo, J. A. Delport, P. Febvre, and C. J. Fourie, "Analysis of a Shielding Approach for Magnetic Field Tolerant SFQ Circuits," *IEEE Trans. Appl. Supercond.*, vol. 27, no. 4, 2017.

[20] U. of California Berkeley, "Berkeley Logic Interchange Format (BLIF)." [Online]. Available: http://www.cs.columbia.edu/~cs6861/sis/blif/index.html

[21] Berkeley, "ABC." [Online]. Available: https://people.eecs.berkeley.edu/{~}alanmi/abc/

[22] C. Wolf, "Yosys." [Online]. Available: http://www.clifford.at/yosys/

[23] M. C. Kim, D. J. Lee, and I. L. Markov, "SimPL: An effective placement algorithm," *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 31, no. 1, pp. 50–60, 2012.

[24] C. J. Fourie, K. Jackman, M. M. Botha, S. Razmkhah, P. Febvre, C. L. Ayala, Q. Xu, N. Yoshikawa, E. Patrick, M. Law, Y. Wang, M. Annavaram, P. Beerel, S. Gupta, S. Nazarian, and M. Pedram, "ColdFlux Superconducting EDA and TCAD Tools Project: Overview and Progress," *IEEE Trans. Appl. Supercond.*, vol. 29, no. 5, 2019.

[25] T. Edwards, "qRouter." [Online]. Available: http://opencircuitdesign.com/qrouter/

[26] C. J. Fourie, "TimEx." [Online]. Available: https://github.com/sunmagnetics/TimEx

[27] S. Williams, "Icarus Verilog." [Online]. Available: http://iverilog.icarus.com/

[28] W. R. Inc, "WRspice." [Online]. Available: http://wrcad.com/wrspice.html

[29] E. S. Fang, "JSIM." [Online]. Available: https://github.com/coldlogix/jsim

[30] P. Shevchenko and T. Van Duzer, "PSCAN." [Online]. Available: http://pscan2sim.org/

[31] J. A. Delport, K. Jackman, P. l. Roux, and C. J. Fourie, "Josim—superconductor spice simulator," *IEEE Transactions on Applied Superconductivity*, vol. 29, no. 5, pp. 1–5, 2019.

[32] "The DOT Language." [Online]. Available: https://www.graphviz.org/doc/info/lang.html

[33] "Graphviz." [Online]. Available: https://graphviz.org/

[34] L. Schindler, "RSFQlib." [Online]. Available: https://github.com/sunmagnetics/RSFQlib

[35] ——, "The Development and Characterisation of an RSFQ Cell Library for Layout Synthesis," Ph.D. dissertation, Stellenbosch University, 2020.

[36] T. Preston-Werner, "TOML." [Online]. Available: https://toml.io/en/

[37] J. Delport, "JoSIM." [Online]. Available: https://github.com/JoeyDelp/JoSIM

[38] J. de Villiers, "Die2Sim." [Online]. Available: https://github.com/judefdiv/Die2Sim

[39] P. Jupyter, "Jupyter." [Online]. Available: https://jupyter.org/

[40] J. de Villiers and H. Herbst, "GDScpp." [Online]. Available: https://github.com/judefdiv/gdscpp

A Cadence Library File for ABC

```
GATE NOTT   2   out = !in;
  PIN * UNKNOWN 1 999 1 1 1 1
GATE AND2T  2   out = in1 * in2;
  PIN * UNKNOWN 1 999 1 1 1 1
GATE OR2T   2   out = (in1 + in2);
  PIN * UNKNOWN 1 999 1 1 1 1
GATE XOR2T  2   out = ( (in1 * !in2) + (!in1 * in2) );
  PIN * UNKNOWN 1 999 1 1 1 1
```

B BLIF File created with ABC for a Full Adder

```
# Benchmark "FullAdder" written by ABC on Tue Jul  7 08:42:25
   2020
.model FullAdder
.inputs a b cin
.outputs cout sum
.gate AND2T in1=b in2=a out=new_n6_
.gate XOR2T in1=b in2=a out=new_n7_
.gate AND2T in1=new_n7_ in2=cin out=new_n8_
.gate OR2T  in1=new_n8_ in2=new_n6_ out=cout
.gate XOR2T in1=new_n7_ in2=cin out=sum
.end
```

C Dot File for a Full Adder

```
digraph FullAdder {
   a [shape=box, color=red];
   b [shape=box, color=red];
   cin [shape=box, color=red];
   cout [shape=box, color=blue];
   sum [shape=box, color=blue];
   a -> AND2T_0;
   a -> XOR2T_1;
   b -> AND2T_0;
   b -> XOR2T_1;
   cin -> AND2T_2;
   cin -> XOR2T_4;
   AND2T_0 -> OR2T_3;
   XOR2T_1 -> AND2T_2;
   XOR2T_1 -> XOR2T_4;
   AND2T_2 -> OR2T_3;
   OR2T_3 -> cout;
   XOR2T_4 -> sum;
}
```

D Testing an Unconnected Output of a Splitter Gate

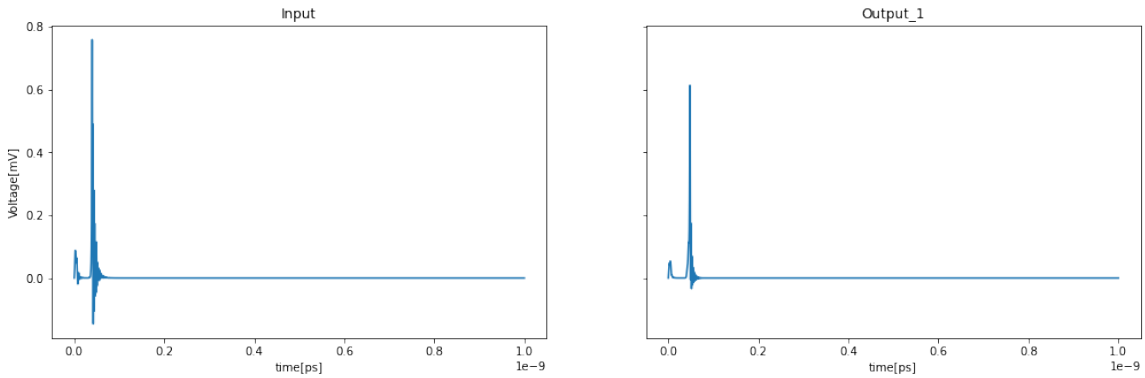

Figure D.1: JoSIM results of a splitter with an unconnected output pin

```
* TESTING THE EFFECTS OF UNCOMMNECT(GROUNDED) OUTPUT OF A
   SPLITTER GATE
* Imported gates of LSmitll_ptltx, LSmitllSPLITT, LSMITLLDCSFQ
*   from https://github.com/sunmagnetics/RSFQlib
Xsplitter1  LSmitllSPLITT  in1  out1  0
T1  net3  0  in1  0  LOSSLESS  Z0=5  TD=1.0p
Iin1          0 net1 pwl(0 0 30p 0 35p 600u 40p 0)
XDCSFQin1     LSMITLLDCSFQ net1 net2
XJTLin1       LSmitll_ptltx  net2 net3
Rout1 out1 0 5
.tran 0.25p 1000p 0 0.25p
.print NODEV net3 0
.print NODEV in1 0
.print NODEV out1 0
.end
```

Listing D.1: JoSIM file used to run the simulation.

E Custom File Describing the Gate Attributes

```
[ALL_GATES]
  port_size   = [4.5, 4.5]        # port(pin) size [x, y]
  ptl_width   = 4.5               # The width of the PLT
     track/route
  via_size    = [4.5, 4.5]        # PTL via size [x, y]
  layer_names = ["metal1", "metal2"]

[JTLT]
  size        = [40, 70]
  origin      = [0, 0]
  pins        = ["in", "out"]
  P1          = [5, 55]           # IN - top left
  P2          = [35, 15]          # OUT - bottom right
  gate_delay = 3.7

[DFF]
  size        = [80, 70]
  origin      = [0, 0]
  pins        = ["clk", "in", "out"]
  P1          = [15, 55]          # CLK - top left
  P2          = [15, 15]          # IN  - bottom left
  P3          = [65, 15]          # OUT - bottom right
  gate_delay = 10.3

[AND2T]
  size        = [100, 70]
  origin      = [0, 0]
  pins        = ["in", "in", "clk", "out"]
  P1          = [15, 15]          # IN1 - bottom left
  P2          = [85, 55]          # IN2 - top right
  P3          = [15, 55]          # CLK - top left
  P4          = [85, 15]          # OUT - bottom right
  gate_delay = 7.0
```

```
[XOR2T]
  size       = [100, 70]
  origin     = [0, 0]
  pins       = ["in", "in", "clk", "out"]
  P1         = [15, 55]          # IN1 - bottom left
  P2         = [15, 15]          # IN2 - top left
  P3         = [85, 55]          # CLK - top right
  P4         = [85, 15]          # OUT - bottom right
  gate_delay = 5.2

[PAD]
  size       = [30, 30]
  origin     = [0, 0]
  pins       = ["in_out"]
  P1         = [15, 15]
  gate_delay = 0
```

Listing E.1: Extract of the custom TOML file describing the gates

F The Different Layout Alignments

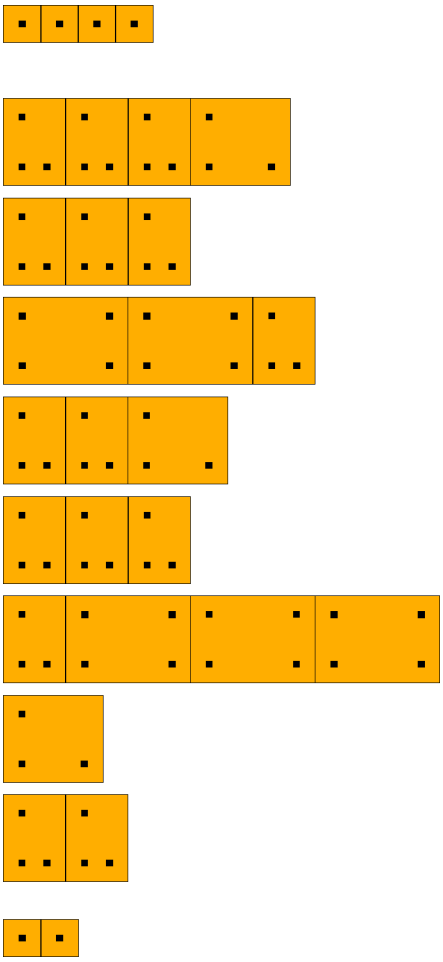

Figure F.1: Flushed left layout of a full adder circuit

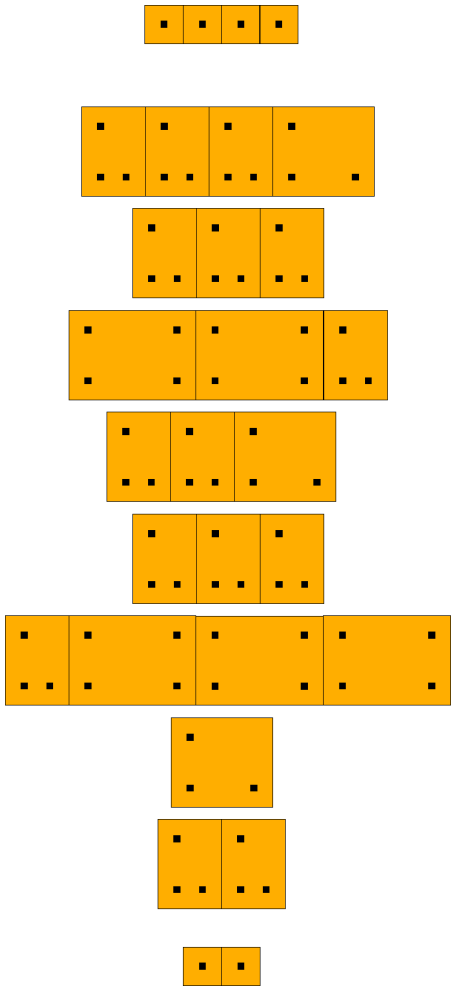

Figure F.2: Centre aligned layout of a full adder circuit

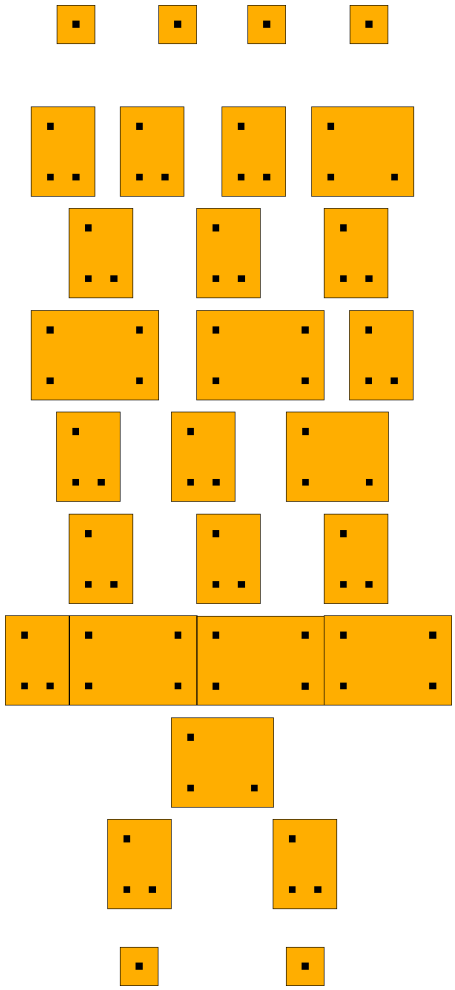

Figure F.3: Centred justified layout of a full adder circuit

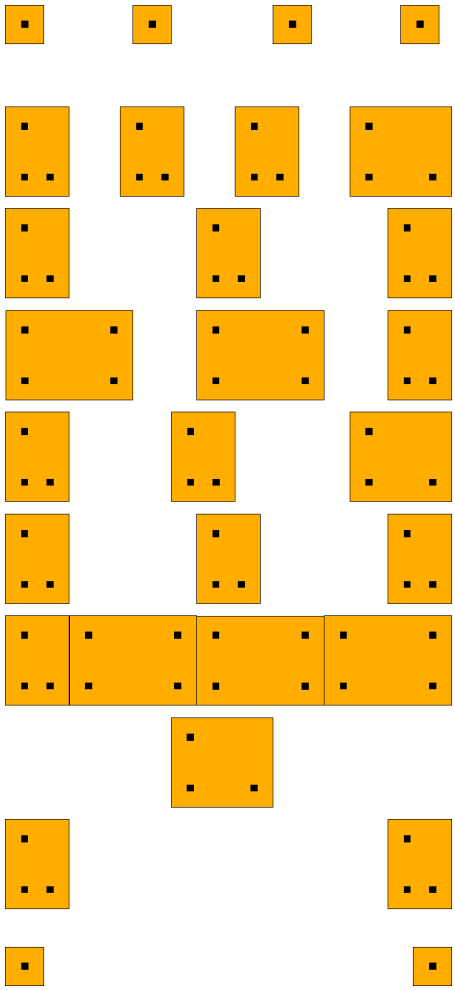

Figure F.4: Full justified layout of a full adder circuit

G Complete Full Adder Circuit

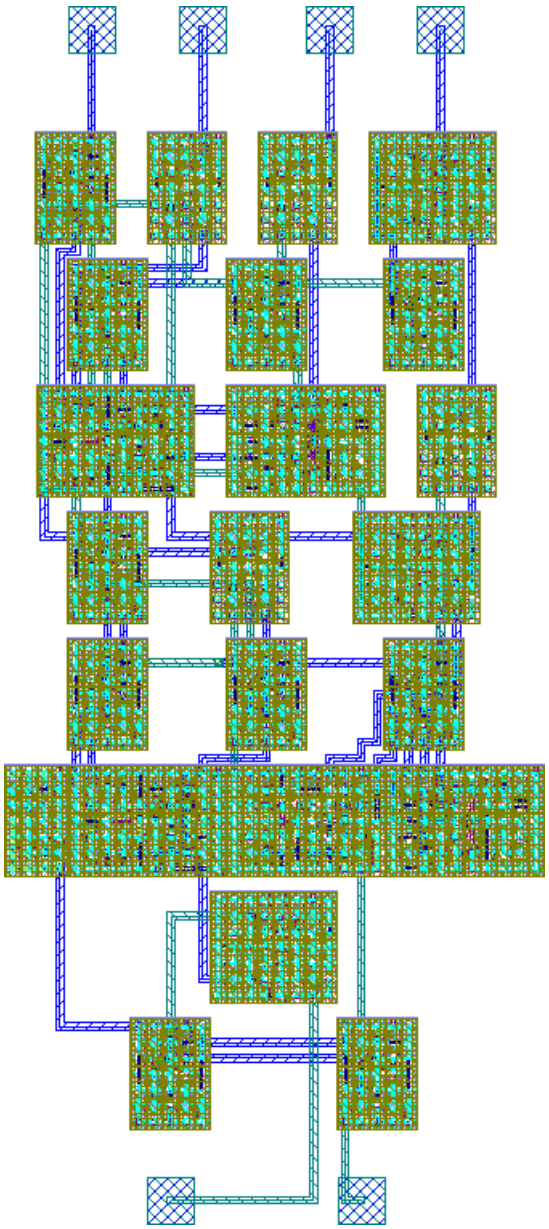

Figure G.1: Complete full adder circuit without fill or biasing

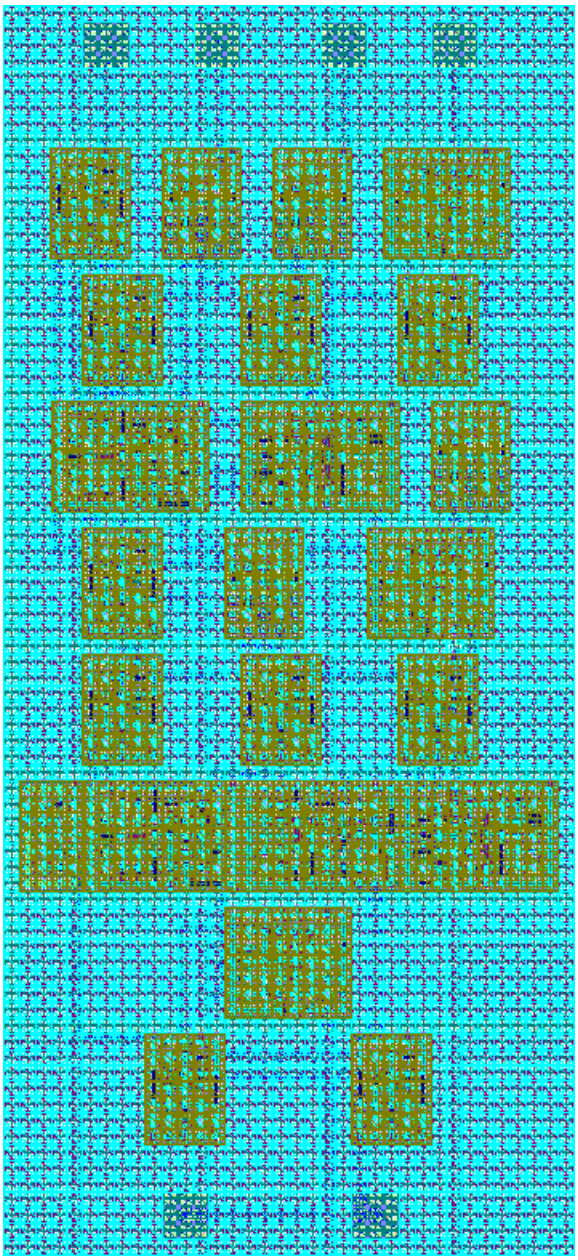

Figure G.2: Complete full adder circuit with fill